TREBALL FINAL DE GRAU

TÍTOL: Sistema d'aparcament intel·ligent basat en F2C (Fog to Cloud)

AUTOR: SALVAT RIBÉ, NIL

DATA DE PRESENTACIÓ: 10, 2019

COGNOMS: SALVAT RIBÉ **NOM: NIL**

TITULACIÓ: Grau en Enginyeria Informàtica

PLA:

DIRECTOR: Sergi Sánchez López

DEPARTAMENT: AC – Departament d'Arquitectura de computadors

QUALIFICACIÓ DEL TFG

TRIBUNAL

PRESIDENT **SECRETARI** **VOCAL**

DATA DE LECTURA:

Aquest Projecte té en compte aspectes mediambientals: □ Sí □ No

"Agrair al meu tutor Sergi Sánchez i al membre del CRAAX Alejandro Jurnet per la seva ajuda i guia al llarg de la realització d'aquest projecte. Agrair també a la meva família per donar-me el suport necessari des del primer fins l'últim dia."

Agraïments

RESUM

La revolució tecnològica en les últimes dècades és innegable, i ha suposat un gran canvi en la societat. La tecnologia forma part del nostre dia a dia i cada vegada tenim més dispositius (i més varietat) connectats a la gran xarxa Internet. Alguns exemples que trobem són sensors, polseres intel·ligents, *smartphones*, inclús neveres i rentadores que fan que els recursos i el tràfic generat a la xarxa sigui d'un abast immens.

És per aquest motiu que neixen els conceptes de *IoT* (Internet of Things o Internet de les coses) i el de *Big Data*. Experts en la matèria calculen que a l'any 2025 hi haurà més de 75 bilions de dispositius connectats a Internet [0].

Pel mateix motiu també neix el concepte de *F2C* (Fog to Cloud), que proposa apropar la tradicional capa cloud (núvol) a on les dades són generades pels dispositius, de manera que es redueix el tràfic que s'envia al cloud i ens aporta nombrosos beneficis, com per exemple poder oferir serveis en temps real.

Per altra banda, les ciutats estan en continu creixement i es necessiten serveis que simplifiquin la vida als ciutadans i al conjunt de la ciutat, partint com no podia ser d'altra manera de la tecnologia.

Si ajuntem els paràgrafs anteriors, ens trobem amb el concepte de "*Smart city*" (ciutat intel·ligent). És un concepte relativament nou, que està desenvolupant-se al llarg del món. Actualment ja trobem *Smart cities* que compten amb serveis per millorar els serveis públics i altres aspectes de la ciutat.

En aquest projecte es proposa un servei d'aparcament intel·ligent amb l'objectiu d'oferir a les ciutats una manera més simple i eficient d'aparcar, partint d'una estructura F2C determinada.

També es proposa un model que utilitza la capacitat de processament dels vehicles aparcats a un pàrquing intel·ligent per tal de poder executar serveis/programes de forma distribuïda, aprofitant així els recursos (que no s'estan aprofitant) mentre els vehicles estan aparcats.

Paraules clau: Smart city, IoT, Big Data, F2C, servei, aparcament intel·ligent, compartició de recursos, edge computing

ABSTRACT

The technological revolution in the last decades in undeniable, and it has supposed a big change in the society. Technology is part of our day to day life and every time we have more devices (and more variety) connected to the large Internet network. Some of the examples that we can found are sensors, smart wearables, smartphones, even refrigerators and washing machines that make the resources and traffic generated on the network be of immense supply.

For this reason we find the concepts of IoT (Internet of Things) and Big Data. Experts estimate that in 2015 there will be more than 75 billion devices connected to the Internet [0].

For the same reason, the concept of F2C (Fog to Cloud) was born. This concept proposes to bring closer the traditional cloud layer to the data generated by devices, in order to reduce the traffic that is sent to the cloud and this brings us numerous benefits, as for example offering real time services.
On the other hand, cities are in a continuous growing and they need services that simplify the citizens life and the whole
city, starting from technology as it could not be otherwise.
If we put together the above paragraphs, we find the concept of Smart city. It's a really recent concept that is being developed around the world and we can already find smart cities that count with services to improve the public services and other aspects of the city.
This project proposes implementation techniques of F2C and different services related to smart parking, with the aim of offering the cities a more simple and efficient way to park.
It also proposes a model that uses the capacity of processing from the vehicles parked in a parking area in order to execute services or programs in a distributed way, taking advantage of resources (which are not being used) while the vehicles are parked.

Key words: Smart city, IoT, Big Data, F2C, service, smart parking, resources sharing, edge computing

Sumari

Sumari de figures

Sumari de taules

GLOSSARI DE TERMES

El següent glossari defineix els termes més utilitzats al llarg d'aquest projecte. Alguns d'ells estan en anglès atès que així es tracten en món de la informàtica.

Agent: dispositiu que, al integrar-se / instal·lar-se / col·locar-se dota de certa intel·ligència i capacitat de còmput a un element quotidià de la ciutat. Els agents són capaços de comunicar-se i de rebre peticions de serveis i executar-les. És l'element bàsic de la topologia.

API: Una interfície de programació d'aplicacions (del anglès Application Programming Interface, API) és una interfície que especifica com diferents components de programes informàtics haurien d'interaccionar. En altres paraules, és un conjunt d'indicacions en quant a funcions i procediments, ofert per una biblioteca informàtica per a ser utilitzat per un altre programa per a interactuar amb el programa en qüestió.

Base de dades: és un conjunt de dades organitzades segons una estructura definida i que és accessible des d'un o més programes o aplicacions, de manera que qualsevol d'aquestes dades pot ser extreta del conjunt, actualitzada o esborrada sense afectar ni l'estructura del conjunt ni a les altres dades.

Clúster: Agrupació o associació de màquines independents integrades per mitjà de xarxes d'interconnexió per obtenir un sistema coordinat, capaç de processar una càrrega.

F2C (Fog to Cloud): Arquitectura de xarxa que proposa acostar la tradicional capa cloud als extrems de la xarxa on les dades són generades, mitjançant la incorporació d'una capa anomenada fog, per tal de que els nodes d'aquesta nova capa duguin a terme càlculs, emmagatzematge, entre d'altres.

Aparcament intel·ligent: és una estrategia d'aparcament que combina la tecnologia i la innovació humana per tal d'utilitzar els menors recursos possibles (gasolina, temps i espai) per aconseguir un aparcament més ràpid, fàcil i eficient.

Framework: és un entorn de treball que incorpora funcions i mòduls que serveixen de base i faciliten el desenvolupament d'aplicacions.

Front-end: És la part d'una aplicació que visualitzen els usuaris i interactuen directament amb ella.

IoT: l'Internet de les coses (de l'anglès, Internet of Things) és un concepte que es refereix a l'extensió d'Internet als objectes quotidians (coses), tals com polseres intel·ligents, càmeres de vídeo, cotxes o fins i tot rentadores. Aquests objectes poden recol·lectar i intercanviar informació per la xarxa i entre ells.

Script: programa informàtic que executa un o més processos definits

Servei: acció o conjunt d'accions d'activitats que busquen satisfer les necessitats d'un client.

Servidor SFTP: és un servidor que utilitza el protocol SFTP (Secure File Transfer Protocol) i que facilita l'accés i la transferència de fitxers entre el servidor que emmagatzema els fitxers i un o més clients que els sol·liciten.

Testbed: banc de proves del CRAAX on es realitzen algunes de les proves de validació del correcte funcionament del model proposat en el projecte.

Topologia: en el context d'aquest projecte anomenarem topologia a la disposició a nivell de xarxa dels agents i enllaços que s'estableixen entre ells dins d'una mateixa ciutat.

Virtualització: és un mecanisme que permet crear i executar diverses màquines virtuals (programari que emula un ordinador i pot executar programes com si ho fos) dins d'una sola màquina física. En el projecte s'utilitzen dos tipus de virtualització: la completa (VirtualBox) i la parcial (Docker). La virtualització ens permet simular una gran quantitat d'agents.

CAPÍTOL 1 - Introducció i objectius del projecte

1.1 Motivació

Avui en dia és molt freqüent veure en les grans ciutats com les persones recorren grans distàncies buscant un lloc on aparcar el seu vehicle, gastant així més diners i temps del necessari. Fer més eficient la mobilitat és un aspecte d'alta importància en qualsevol ciutat, i sobretot ho és en les ciutats intel·ligents.

És per aquest motiu que necessitem gestionar i oferir serveis d'aparcament que ens permetin facilitar l'aparcament de vehicles, reduint així el tràfic a la ciutat i la pol·lució generada pels vehicles que circulen buscant aparcament i també estalviant temps i diners als ciutadans.

A més, veiem com a les ciutats existeixen grans pàrquings amb grans quantitats de vehicles estacionats. En un entorn específic de ciutat intel·ligent i partint de la base de que cada vehicle és un node de la ciutat (incorpora un agent), disposem de grans *clústers* o agrupacions de nodes amb uns recursos computacionals que mentres el vehicle està estacionat no s'estan utilitzant i per tant s'estan desaprofitant.

Dit això, perquè no convertir aquestes agrupacions de nodes en "supercomputadors" que ens permetin executar altres serveis i programes externs i al mateix temps el client que cedeix el seu vehicle en surti beneficiat, per exemple, amb algun tipus de descompte?

Davant d'aquestes dues problemàtiques exposades sorgeix la necessitat de poder gestionar aquests aparcaments intel·ligents i que alhora els vehicles serveixin per a satisfer altres necessitats, ja siguin pròpies de la ciutat o d'entitats externes.

1.2 Origen del projecte

La idea i proposició inicial de realització d'aquest projecte sorgeix del grup d'investigació multidisciplinari CRAAX (Centre de Recerca d'Arquitectures Avançades de Xarxes) [1] de l'EPSEVG (Escola Politècnica Superior d'Enginyeria de Vilanova i la Geltrú) [2].

Actualment els membres del CRAAX es troben investigant en el desenvolupament de tecnologies de xarxes i enfocant els seus treballs a un entorn de ciutats intel·ligents: transport intel·ligent, salut electrònica, edificis intel·ligents, entre d'altres.

Tant és així que disposen d'un banc de proves "testbed" que permet simular serveis aplicables a una ciutat real, però evidentment a una escala reduïda. Aquest testbed disposa de semàfors, fanals, automòbils, carreteres, edificis i també una zona d'aparcament, utilitzada en el projecte.

El CRAAX disposa d'un laboratori situat a l'edifici Neàpolis de Vilanova i la Geltrú [3], on juntament amb altres estudiants de grau, màster o doctorat (PhD) porten a terme les seves investigacions i els seus projectes.

1.3 Objectius

Aquest projecte pretén donar suport a les investigacions que s'estan duent a terme per les ciutats intel·ligents, en especial a una àrea d'aparcament intel·ligent, així com aportar idees pròpies per al seu desenvolupament.

L'objectiu principal d'aquest projecte és el disseny i desenvolupament de serveis que ens permetin fer més eficient els aparcaments d'una ciutat intel·ligent.

Com a objectiu secundari trobem el disseny i desenvolupament dels elements de la ciutat intel·ligent que necessitem per a poguer dur a terme els serveis esmentats.

Aquesta àrea d'aparcament intel·ligent contempla dos escenaris:

1. Fer més fàcil, eficient i eficaç l'aparcament de vehicles en una ciutat intel·ligent, oferint un servei d'aparcament intel·ligent.

2. Aprofitament de la capacitat de còmput dels nodes (vehicles) estacionats en un pàrquing per a processar altres serveis o aplicacions de la ciutat o també programes externs.

Llavors, podem desglossar els objectius d'aquest projecte en objectius tècnics i objectius acadèmics, detallats a continuació.

1.3.1 Objectius tècnics

Els objectius que segueixen s'enfoquen en l'aspecte professional, és a dir, el fet de realitzar un projecte complet i funcional i assolir els objectius inicials proposats.

- Disseny i implementació a nivell de software dels agents, així com les comunicacions i pas de missatges entre ells.
- Virtualització d'agents per tal de poder tenir-ne un nombre elevat (simulació d'una ciutat o un pàrquing real).
- Disseny i implementació de serveis relacionats amb l'aparcament intel·ligent.
- Integració de part del projecte amb altres projectes que s'estan desenvolupant paral·lelament a aquest per tal de potenciar algunes de les funcionalitats implementades.
- Realització de les proves pertinents per verificar el correcte funcionament de cada element implementat en el projecte.
- Disseny i implementació d'una aplicació web (front-end) per administrar un pàrquing F2C.
- Disseny i implementació d'una aplicació destinada als nostres clients per tal de que puguin sol·licitar els serveis implementats en el projecte.
- Elecció i aprenentatge autònom de les tecnologies pertinents per a desenvolupar el projecte.

1.3.2 Objectius acadèmics

Els objectius que segueixen s'enfoquen en l'aspecte acadèmic, és a dir, com a estudiant que està finalitzant el grau d'Enginyeria Informàtica.

- Aplicació d'una metodologia àgil per a desenvolupar el projecte.
- Treballar paral·lelament i conjuntament amb altres projectes que estan sent desenvolupats per altres estudiants del mateix grau i enfocats en l'àmbit de ciutats intel·ligents.
- Assentar les bases d'una futura línia de treball en termes d'aparcament intel·ligent i ciutats intel·ligents.
- Redactat de la present memòria i documentació tècnica (funcionament de cada element implementat i manuals per ficar-ho tot en funcionament).
- Aconseguir la resta d'objectius plantejats per tal d'oferir al CRAAX el resultat final del projecte.

1.4 Estructura de la memòria i resum de continguts

A continuació es resumeixen els capítols que constitueixen la memòria del projecte:

- Capítol 1 (Introducció i objectius del projecte): En aquest primer capítol s'explica el perquè necessitem sistemes d'aparcament intel·ligent i també quins són els objectius que es pretenen aconseguir un cop finalitzat el projecte. A més, inclou un breu descripció del contingut de cada capítol de la present memòria.
- Capítol 2 (Planificació del projecte i metodologia emprada): Es descriu la metodologia que s'ha seguit al llarg del desenvolupament del projecte, així com també la planificació general d'aquest.

- Capítol 3 (Estat de l'art): Anàlisi i descripció d'alguns sistemes d'aparcament intel·ligent que estan funcionant avui en dia: característiques, quin valor ens aporten, com funcionen, model de negoci, entre d'altres. També inclou un breu resum d'altres conceptes que estan directament relacionats amb el projecte.

- Capítol 4 (Arquitectura F2C): En aquest capítol s'explica què és el F2C (Fog to Cloud), per a que el necessitem i quins beneficis ens aporta en l'àmbit de ciutats intel·ligents. També es descriuen les utilitats i el valor que aporta al projecte que s'està realitzant. A més, inclou una breu explicació sobre el projecte europeu mF2C.

- Capítol 5 (Anàlisi de requisits): Es contextualitza el projecte i es defineixen quines seran les funcionalitats de cada element que es dissenyaran i implementaran al llarg del projecte.

- Capítol 6 (Disseny del sistema): Explica en detall quines tecnologies s'han decidit emprar i el plantejament sobre com s'implementarà cadascun dels els elements del projecte.

- Capítol 7 (Implementació): Primerament es fa un resum de les tecnologies emprades en la implementació, a més d'una explicació de les eines que s'han utilitzat per al desenvolupament.

  També s'explica com s'ha implementat cadascun dels elements que formen el projecte, incloent els trossos i funcions de codi més rellevants i captures de pantalla del resultat.

- Capítol 8 (Serveis): Aquest capítol detalla els serveis que s'han implementat al projecte i parla dels nostres clients i del model de negoci.

- Inclou diagrames amb els missatges i les comunicacions que s'estableixen en cada pas del servei per tal d'entendre què està passat a cada moment.

- Capítol 9 (Integració amb altres projectes): S'explica en detall quines parts del projecte s'han integrat i com, així com també una explicació dels altres projectes i objectius de la integració. A més, inclou l'explicació del servei d'aparcament conjunt amb un altre equip.

- Capítol 10 (Proves de funcionament): Es realitzen i documenten les proves pertinents per assegurar el correcte funcionament de cadascun dels elements implementats.

- Capítol 11 (Continuïtat del projecte): Plantejament de possibles funcionalitats i aspectes a afegir al projecte realitzat.

- Capítol 12 (Cost del projecte): Estimació dels recursos necessaris (humans i materials) per a completar les activitats del projecte.

- Capítol 13 (Conclusions finals del projecte): Inclou les conclusions obtingudes de la realització del projecte. Per això, es fa una valoració global (objectiva i subjectiva) del projecte realitzat i es comprova que s'hagin assolit els objectius definits al capítol 1.

- Capítol 14 (Referències): Engloba totes les referències mencionades al llarg de la redacció de la present memòria.

- Capítol 15 (Bibliografia): Engloba totes les fonts d'informació (bibliogràfiques i informàtiques) utilitzades per a la realització del projecte.

- Annex 1 (Manuals): Guia d'instal·lació de cadascun dels elements del projecte i posada en marxa d'aquests. També inclou una explicació del funcionament de la llibreria Parallel Python i de la API dels agents implementada en el projecte.

CAPÍTOL 2 – Planificació del projecte i metodologia emprada

Aquest capítol descriu la planificació temporal del projecte sencer, així com la metodologia que s'ha utilitzat durant el desenvolupament d'aquest.

2.1 Metodologia de desenvolupament

Per al desenvolupament d'aquest projecte s'ha utilitzat una metodologia "àgil" anomenada SCRUM. Aquesta metodologia està especialment pensada per a treballs en equip, però també és vàlida per a projectes d'un sol desenvolupador.

SCRUM es basa en fer petits increments del projecte compresos en un període de temps (en aquest cas dues setmanes) que els hi direm sprints o iteracions.

L'objectiu de cada sprint és realitzar un PVM (Producte Viable Mínim) per ensenyar al client. Aquest producte ha de tenir les suficients característiques per a satisfer als clients inicials i que puguin donar retroacció per al futur desenvolupament.

En aquest cas els clients són el propi tutor del projecte (Sergi Sánchez) i un dels membres del CRAAX (Alejandro Jurnet). Aquest últim desenvolupa el rol de client tècnic. El client tècnic ens dóna indicacions més teòriques sobre el projecte, i també ens ha explicat diversos conceptes sobre els agents i topologia.

Llavors, cada sprint consta de:

1. Planificació del sprint. En aquest primer pas es concreta quins són els objectius del sprint i s'especifiquen les tasques/accions que es duran a terme per assolir-los, a partir de les històries d'usuari. Les històries d'usuari són descripcions curtes i esquemàtiques que resumeixen una necessitat concreta del client i proposen una solució que la satisfà. Les històries d'usuari han de ser el més independents entre sí possibles, per tal de poguer paral·lelitzar la implementació del projecte entre diferents membres de l'equip.
   A més, la planificació de cada sprint s'ha de fer tenint en compte la retroacció (*feedback*) que ens ha donat el client a la demo del sprint anterior.
2. Realització per part de l'equip tècnic de les tasques que s'han especificat a la planificació.
3. DEMO (presentació) del producte obtingut de la realització de les tasques al client.

És en aquest últim pas del sprint on el client ens dóna el *feedback* de la feina feta i ens diu que li agradaria veure a la següent presentació.

En el cas de que, en un sprint, per qualsevol motiu, no s'hagin finalitzat totes les tasques previstes, les tasques restants es deixaran pel següent sprint.

Aquesta metodologia compta amb un tauler (també anomenat tauler Kanban) visible per a tot l'equip on podem veure les diferents històries d'usuari i tasques a desenvolupar. Al següent apartat d'aquest capítol s'explica més en detall el seu funcionament i per a que el necessitem.

El gran avantatge que ens aporta aquesta metodologia és que el client ens dóna indicacions cada poc temps (més participació i involucració per part del client) i així si hem de canviar alguna cosa del producte no haurem de desfer gaire, només alguna cosa del sprint anterior que al client no li agradi o vulgui modificar. A més, és una metodologia molt utilitzada en projectes de recerca on els objectius són molt amples.

Atès que en aquest cas el projecte és realitzat per una sola persona, no es tenen en compte cap dels "rols" que apareixen a SCRUM, ni tampoc les reunions diàries amb la resta de l'equip ("dailies") o reunions setmanals ("weeklies"). Tampoc les "retros" (reunions amb l'equip després de cada demo per debatre el sprint).

2.2 Eines de seguiment

Per tal d'administrar les històries d'usuari, les tasques i l'estat de cada sprint s'ha utilitzat *Trello* [4]. Aquesta aplicació web o per a dispositius mòbils ens ofereix un espai de treball en forma de tauler on podem crear les històries d'usuari i les tasques que es duran a terme a cada sprint.

En aquest tauler podem veure quines tasques s'estan realitzant en aquest moment ("DOING"), quines ja s'han fet ("DONE") i quines encara no s'han començat a fer ("TO DO"), així com quin membre de l'equip és el responsable de cada tasca.

Això ens permet tenir una visió molt clara de l'estat actual del sprint (és a dir, si anem endarrerits o avançats segons la planificació i temporització del sprint).

En aquest projecte el tauler també és visible per part dels clients, per això també s'ha utilitzat *Trello* per concretar les reunions amb els clients enmig de cada sprint, així com per la compartició de fitxers.

A la figura 1 podem veure una captura de pantalla del tauler, presa enmig del sprint 8.

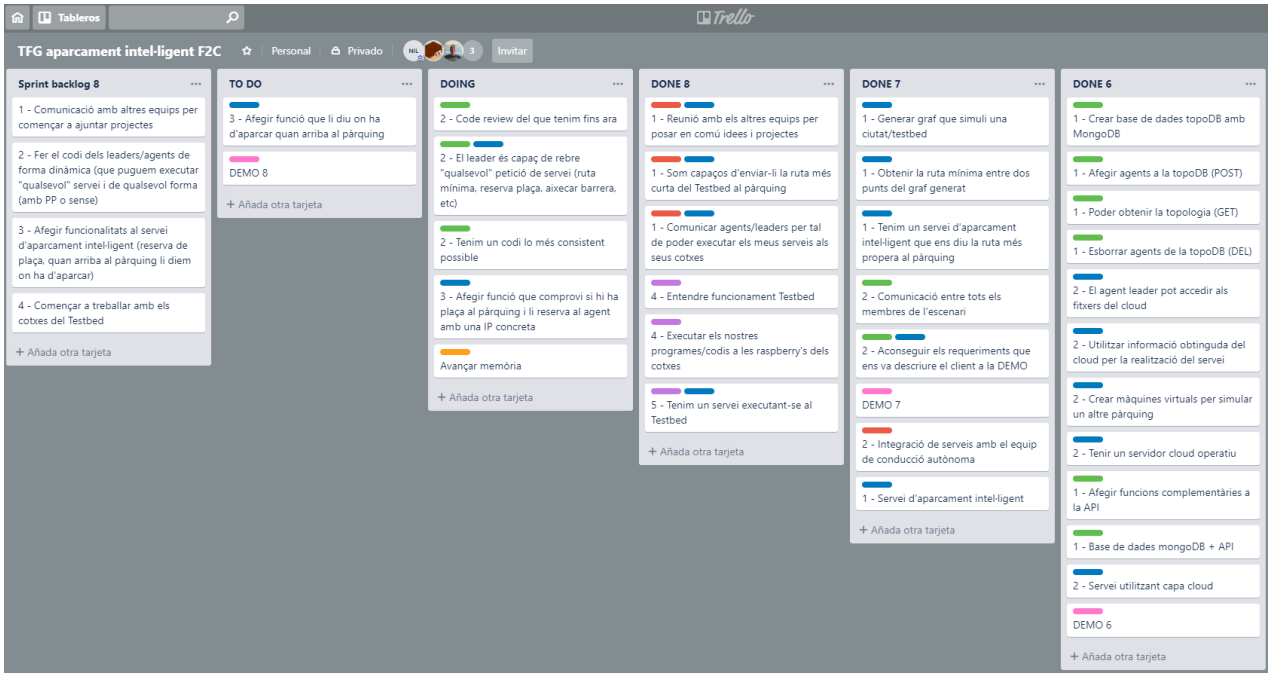


Figura 1: tauler virtual de Trello

2.3 Mètodes de validació

Com ja s'ha comentat en aquest capítol, al final de cada sprint s'ha fet una demo amb els clients on s'ha presentat el resultat obtingut al sprint. Aquest ha sigut el mètode principal de validació dels avanços que s'han anat produïnt al llarg del projecte.

El seguiment i validació del projecte ha sigut doncs continu (cada dues setmanes com a molt) per part del tutor i hi han hagut reunions enmig de cada sprint, tant amb el tutor com amb el client.

A més, s'han realitzat proves de funcionament al final de cada història d'usuari completada per tal de validar el resultat obtingut de les tasques realitzades.

2.4 Resum temporal del projecte

Resum del projecte	
Data d'inici	7/1/2019
Data final	2/10/2019
Data de lectura	24/10/2019
Duració iteracions / sprints	2 setmanes
Càrrega de treball *	18 hores setmanals
Hores totals previstes *	540
Hores finals	576

Taula 1: resum del projecte

* El present treball final de grau correspon a 18 crèdits, que segons la UPC 1 crèdit són 30 hores de dedicació.

Veiem (taula 1) com hi ha hagut una desviació respecte a la planificació inicial de les hores que es dedicarien al projecte. Això es deu a la complexitat d'enfrentar-se per primera vegada a un projecte d'aquestes característiques i a altres aspectes tals com la integració de part del projecte amb altres projectes desenvolupats paral·lelament a aquest.

2.5 Informe de temps del projecte

La següent taula mostra les dates de cada sprint: la primera data correspon a l'inici del sprint, mentres que la segona fa referència al dia de la demo. Notem que hi ha algun sprint que dura una més de dues setmanes degut a les vacances de setmana santa o d'estiu.

Resum dels sprints o iteracions	
Sprint 0	7/1/2019 – 16/1/2019
Sprint 1	16/1/2019 – 30/1/2019
Sprint 2	30/1/2019 – 13/2/2019
Sprint 3	13/2/2019 – 27/2/2019
Sprint 4	27/2/2019 – 13/3/2019
Sprint 5	13/3/2019 – 27/3/2019
Sprint 6	27/3/2019 – 10/4/2019
Sprint 7	10/4/2019 – 2/5/2019
Sprint 8	2/5/2019 – 15/5/2019
Sprint 9	15/5/2019 – 29/5/2019
Sprint 10	29/5/2019 – 12/6/2019
Sprint 11	12/6/2019 – 3/7/2019
Sprint 12	3/7/2019 – 17/7/2019
Sprint 13	17/7/2019 – 4/9/2019
Sprint 14	4/9/2019 – 18/9/2019
Sprint 15	18/9/2019 – 2/10/2019

Taula 2: resum temporal dels sprints

2.6 Resum de les iteracions / sprints

2.6.1 Sprint 0 (introductori)

En aquest sprint inicial s'ha investigat sobre smart cities i també sobre alguns projectes que s'estan portant a terme avui en dia sobre smart pàrquings. A la demo, s'ha presentat l'estat de l'art. L'objectiu d'aquest primer sprint no és més que endinsar-nos en la matèria.

2.6.2 Sprint 1

S'ha començat a desenvolupar l'estructura general del pàrquing i com estarà lligat amb F2C. En aquest sprint s'investiga a fons el model F2C (aventatges, perquè el necessitem). També es fa un primer redactat de la memòria amb l'estat de l'art i F2C.

2.6.3 Sprint 2

Investigació de quines llibreries o aplicacions ens poden servir per executar un programa de forma distribuïda en diverses màquines. El client ens explica què són els agents i la topologia que utilitzarem en el projecte.

2.6.4 Sprint 3

Primera execució d'un programa de forma distribuïda amb màquines virtuals (que simulen els vehicles estacionats al pàrquing). En aquest sprint també s'investiguen altres llibreries i eines que ens poden servir per fer l'execució, a part de la triada (Parallel Python). El client ens explica amb detall els agents (mòduls, API).

2.6.5 Sprint 4

Demostració de que el programa s'executa de forma distribuïda amb Parallel Python. També s'ha treballat en la topologia del pàrquing a nivell d'agents, i s'ha començat a implementar els agents.

A més, s'ha redactat una primera versió d'un glossari conjunt amb altres equips que porten a terme projectes enfocats en ciutats intel·ligents.

2.6.6 Sprint 5

Primera versió d'un servei real complet (acció que causa el servei, petició de servei, execució i resultat). El servei en qüestió tracta d'una simulació d'un aparcament intel·ligent, on se li envia la ruta del pàrquing més proper a un cotxe que fa la petició del servei. Es revisa glossari conjunt amb els altres equips de treball per tal d'obtenir-ne una versió definitiva.

2.6.7 Sprint 6

En aquest sprint es treballa amb dues zones de la ciutat (al tenir dues zones intervé l'agent cloud per comunicar els leaders de cada zona). A la demo, s'ensenya al client el flux de les dades i comunicacions que es realitzen a l'hora d'executar un servei en el cas de que tinguem més d'un pàrquing o més d'una zona.

2.6.8 Sprint 7

Es millora el model F2C que ja teniem per tal de complir amb els requeriments que ens han posat els clients a la demo del sprint anterior. Es segueix treballant en la API i implementació dels agents. A la demo es presenta el model millorat.

2.6.9 Sprint 8

En el sprint 8 continuem millorant la classe agent i els seus mòduls. El client ens explica la necessitat de tenir un panell d'administració per al pàrquing.

2.6.10 Sprint 9

Es defineixen les funcionalitats i a dissenyar el panell front-end. A la demo es presenta una primera versió del panell amb les funcionalitats i tecnologies escollides.

2.6.11 Sprint 10

Principalment es treballa en la implementació del panell d'administració, sense deixar de banda els agents i els serveis. A la demo es presenta una versió bastant millorada del panell amb una primera versió de tots els mòduls que el componen i el seu funcionament.

2.6.12 Sprint 11

Es continua treballant en el panell d'administració, en especial la part de generalització i dibuix del pàrquing.

A banda del panell d'administració, es fa la virtualització d'agents amb Docker.

2.6.13 Sprint 12

Primera integració amb l'equip que està fent el projecte MIAU. En aquesta integració fem que els seus agents executin el servei del nostre projecte d'aparcament.

Es treballa i millora el servei de computació en el seu mòdul al front-end.

Reunions amb l'equip de conducció autònoma per a fer el disseny del servei d'aparcament conjunt al testbed.

2.6.14 Sprint 13

Aquest sprint s'enfoca en la redacció de la memòria del projecte, tot i que també es finalitza la implementació front-end, amb tots els mòduls operatius i millores estètiques. A més, s'ha fet el servei d'aparcament (al testbed del CRAAX) juntament amb l'equip de conducció autònoma.

2.6.15 Sprint 14

Revisió general del projecte i finalització de la memòria. A la demo s'entrega la versió final de la memòria i una demostració completa de tot el projecte.

En aquest sprint també s'ha fet una revisió de tot el codi (*code review*) per tal de netejar-lo una mica de cara a l'entrega final (posar últims comentaris, esborrar funcions que no s'utilitzen, entre d'altres).

2.6.16 Sprint 15

En el darrer sprint del projecte s'han corregit alguns aspectes de la memòria a partir de la retroacció donada per part dels clients. També s'ha fet la guia d'instal·lació i posada en marxa del projecte.

A la demo s'entrega la guia al client i aquest completa amb èxit la instal·lació i posada en marxa del projecte al seu ordinador.

CAPÍTOL 3 - Estat de l'art

El present estat de l'art abarca alguns dels conceptes més importants a l'hora de parlar de ciutats intel·ligents, així com també un anàlisi d'algunes de les empreses i les solucions que actualment estan oferint en l'àmbit d'aparcament intel·ligent i que d'alguna manera s'assemblen als objectius d'aquest projecte.

3.1 Internet of Things

Per a poder entendre la importància de les smart cities primer hem de saber sobre la revolució tecnològica que porta amb ella mateixa el IoT, un concepte cada vegada més important.

Es coneix com a Internet of Things (Internet de les coses) la extensió d'Internet als objectes quotidians (coses), tals com polseres intel·ligents, càmeres de vídeo, cotxes o fins i tot rentadores. Aquests objectes poden recol·lectar i intercanviar informació per la xarxa i entre ells.

L'empresa analista Gartner [5] preveu que per al proper any 2025 hi hagin més de 75 bilions de dispositius connectats a Internet (figura 2).

L'Internet de les coses ofereix aplicacions molt diverses tals com la domòtica, robòtica, transport intel·ligent / autònom, seguretat (càmeres de vigilància), entre moltes altres.

Aquesta extensió és possible gràcies a alguns motius com el decreixent cost de la connectivitat, a que més dispositius són creats amb Wi-Fi i sensors, i el abaratiment dels costos de la tecnologia en general.

En les ciutats intel·ligents, l'IoT s'implanta mitjançant sensors, donant importància als serveis públics tals com el transport, la il·luminació, recollida de residus, edificis intel·ligents, entre d'altres. Amb això es pretén aconseguir un maneig eficient que redueixi la despesa pública. A més, amb la presència de sensors i altres dispositius IoT permet supervisar i controlar les instal·lacions a distància, així com recopilar dades útils per a l'adopció de mesures sobre l'ús de l'energia o el manteniment i gestió de residus.

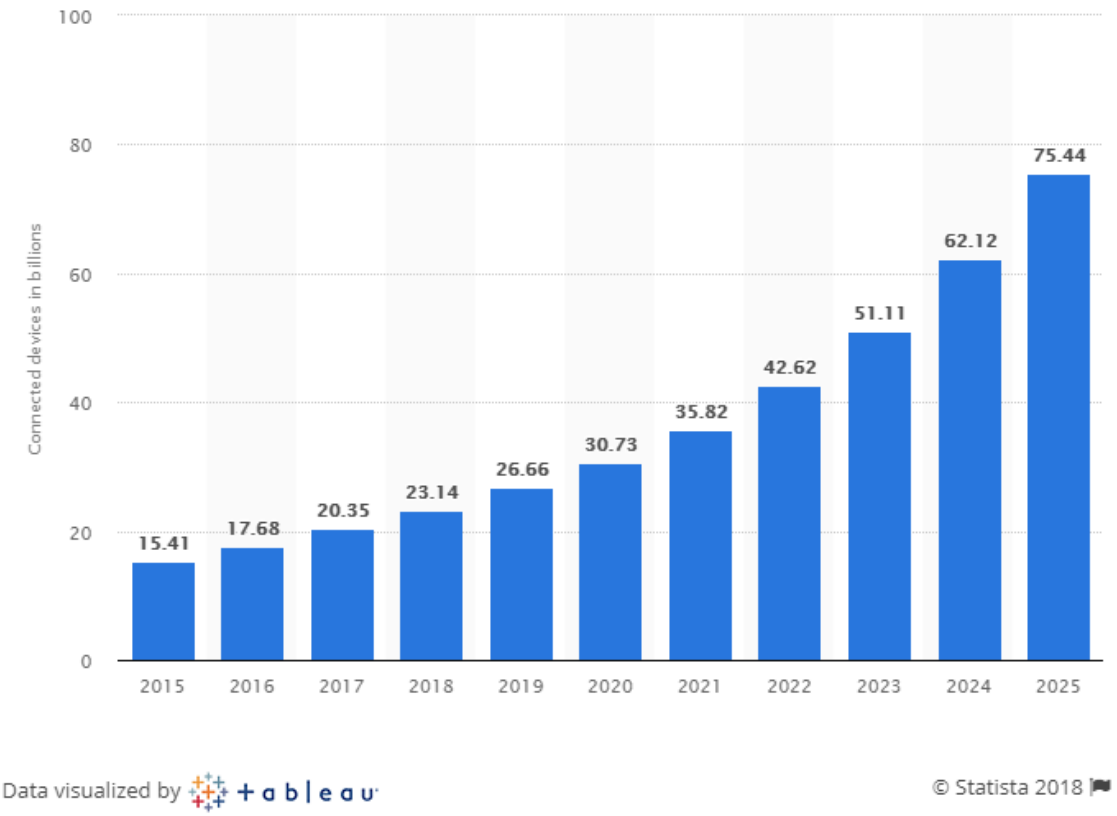


Figura 2: evolució del nombre de dispositius connectats a Internet

3.2 Big Data

El terme Big Data s'utilitza per a fer referència a grans volums de dades que actualment existeixen arreu del món, generades a partir de fonts molt diferents (entre elles els dispositius IoT) i per tant amb diferent format.

Tanmateix fa referència a **que es fa** amb aquestes dades. Són moltes les empreses que estan dedicant grans esforços al tractament d'aquests enormes conjunts de dades per tal d'extreure'n alguna conclusió.

No obstant, al tractar-se d'un volum inabastable de dades, no som capaços d'analitzar-les totes. Segons el diari britànic "The Guardian" [6], menys del 1% de totes les dades generades van ser tractades a l'any 2012.

Es necessiten doncs mètodes diferents als convencionals per a emmagatzemar-les i processar-les de manera eficient. Alguns d'aquests mètodes poden ser les bases de dades no estructurades (també anomenades no SQL) o altres sistemes com Hadoop [7] o Spark [8].

En aquest projecte el Big Data no té massa importància ja que les proves que es faran seran petites, però si apliquessim el mateix a una ciutat real sí que seria molt rellevant.

La figura 3 mostra algunes de les dades que actualment es produeixen cada minut a Internet:

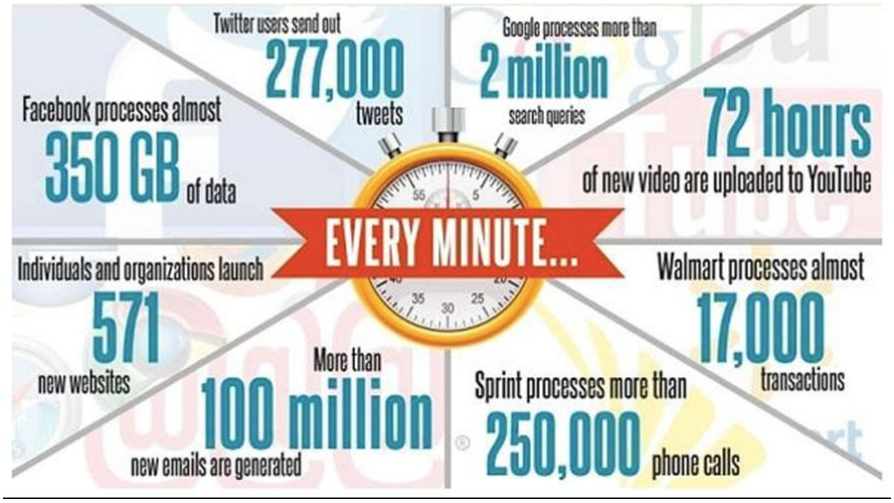


Figura 3: algunes dades del Big Data

3.3 Smart Cities

Es preveu que a l'any 2050 un 70% de la població visqui en ciutats. A Espanya, es calcula que actualment el 80% de les persones viu en ciutats. Aquestes necessiten obligatòriament donar cabuda a tants milions de persones, amb les millors prestacions i recursos possibles.

El concepte "Smart City" (en català ciutat intel·ligent) és un concepte emergent i per tant no té una única definició. Tot i així, podriem definir una Smart City com una ciutat o àrea urbana que utilitza dades i informació aportada per diferents dispositius (sensors, dispositius IoT…) per poder gestionar i oferir serveis als seus habitants, així com millorar la seva qualitat de vida mitjançant solucions tecnològiques.

Les ciutats intel·ligents han de ser capaces d'utilitzar la tecnologia de la informació i la comunicació (TIC) amb l'objectiu de crear millors infraestructures per als ciutadans. Des de transport públic, passant per a l'estalviament energètic, sostenibilitat o eficiència en tots els seus aspectes. La figura 4 mostra alguns dels àmbits més importants a tractar per les Smart cities.

El compromís mediambiental és també un dels principals pilars d'una ciutat intel·ligent. Això implica que en les ciutats intel·ligents s'hauran d'aplicar les noves tecnologies per tal de reduir les emissions de CO2 i partícules perilloses en l'aire, fer una gestió més eficient i controlada en l'aigua amb l'objectiu de consumir-ne menys o minimitzar la generació de residus i millorar la seva recollida i tractament. Aquests són només alguns dels exemples on les ciutats intel·ligents poden ser de gran ajuda.

Amb els sistemes d'aparcament intel·ligent (que ja trobem avui en dia en algunes ciutats) podem aconseguir que hi hagi menys tràfic a la ciutat, reduint així la pol·lució generada pels vehicles que circulen buscant un lloc per aparcar i estalviant temps i diners als ciutadans.

Figura 4: aspectes més importants a l'hora de parlar de ciutats intel·ligents

3.4 Smart cities d'avui en dia

A continuació es detallen alguns dels projectes i empreses que actualment estan desenvolupant i ofereint serveis d'aparcament en ciutats intel·ligents.

3.4.1 Smart Santander

Smart Santander [9] és un projecte desarrollat a la ciutat de Santander i que té com a objectius dissenyar, desplegar i validar a Santander i al seu entorn una plataforma formada per sensors, actuadors, càmeres i pantalles per oferir informació útil als ciutadans.

Aquest projecte ha desplegat més de 20.000 dispositius a Santander i al seu entorn (sensors, repetidors…). A més, 1125 Waspmotes [10] (dispositius similars als Arduino [11]) han sigut desplegats per tal de monitoritzar diferents paràmetres tals com el soroll, la temperatura, nivells de lluminositat, CO2 i també places d'aparcament lliures al llarg de la ciutat.

Un dels serveis que s'ofereixen i que està directament relacionat amb aquest projecte és el control d'ocupació i ús de places de pàrquing reservades a persones amb mobilitat reduïda.

A més, en relació a serveis d'aparcament, també s'han col·locat panells indicatius que guien als conductors cap a places d'aparcament disponibles. Aquests panells (figura 5) estan situats als carrers principals i n'hi ha un total de 10.

Figura 5: panell indicatiu de places d'aparcament disponibles, a la ciutat de Santander

La infraestructura desplegada a la ciutat també ofereix un banc de proves que proporciona a la comunitat investigadora una plataforma per a la experimentació a gran escala i evaluació dels conceptes IoT en condicions reals.

3.4.2 Pàrquing Callao

El pàrquing robotitzat Callao [12] està situat a la Gran Via de Madrid. Consta de 9 plantes amb un total de 320 places d'aparcament.

Pots reservar la teva plaça des de la seva pàgina web o bé d'algunes app's d'aparcament indicades al seu web. Les reserves es poden fer per hores, dies o setmanes i el pagament el pots fer per avançat o bé al marxar.

Quan arribes al pàrquing, t'enregistren i deixes el teu vehicle a una plataforma robotitzada que l'aparca de manera automàtica. D'igual manera, a la sortida els clients recullen el vehicle directament des de la plataforma, com es mostra a la figura 6:

Figura 6: plataforma robotitzada del pàrquing Callao

La seguretat en aquest pàrquing és molt elevada ja que cap persona està autoritzada a entrar a l'espai on es troben els vehicles estacionats, evitant així danys per vandalisme, robatoris o frecs entre vehicles causats per altres usuaris.

Disposa de Wi-Fi en tot el perímetre del pàrquing i està adaptat per a minusvàlids i persones amb mobilitat reduïda.

3.4.3 Mobypark

Mobypark [13] és una plataforma que permet compartir i optimitzar les places de garatge que hi ha a la ciutat.

Els propietaris de places de garatge individuals, pero també hotels, empreses dedicades a l'aparcament i edificis poden llogar les seves places de garatge quan no estan sent utilitzades. Això permet als conductors accedir a un major nombre de places de garatge que en un principi no estarien accesibles al públic.

En aquest cas, doncs, no tenim un sol pàrquing, sinó que en tenim diversos en una mateixa ciutat. Com en l'anterior cas, podem reservar plaça via pàgina web (figura 7) o des de l'aplicació mòbil.

Mobypark permet llogar per hora, dia, setmana o qualsevol duració que el propietari de la plaça estableixi. Aquesta flexibilitat permet que conductors que volen aparcar només uns dies puguin accedir a places de garatge que usualment estan llogades mensualment.

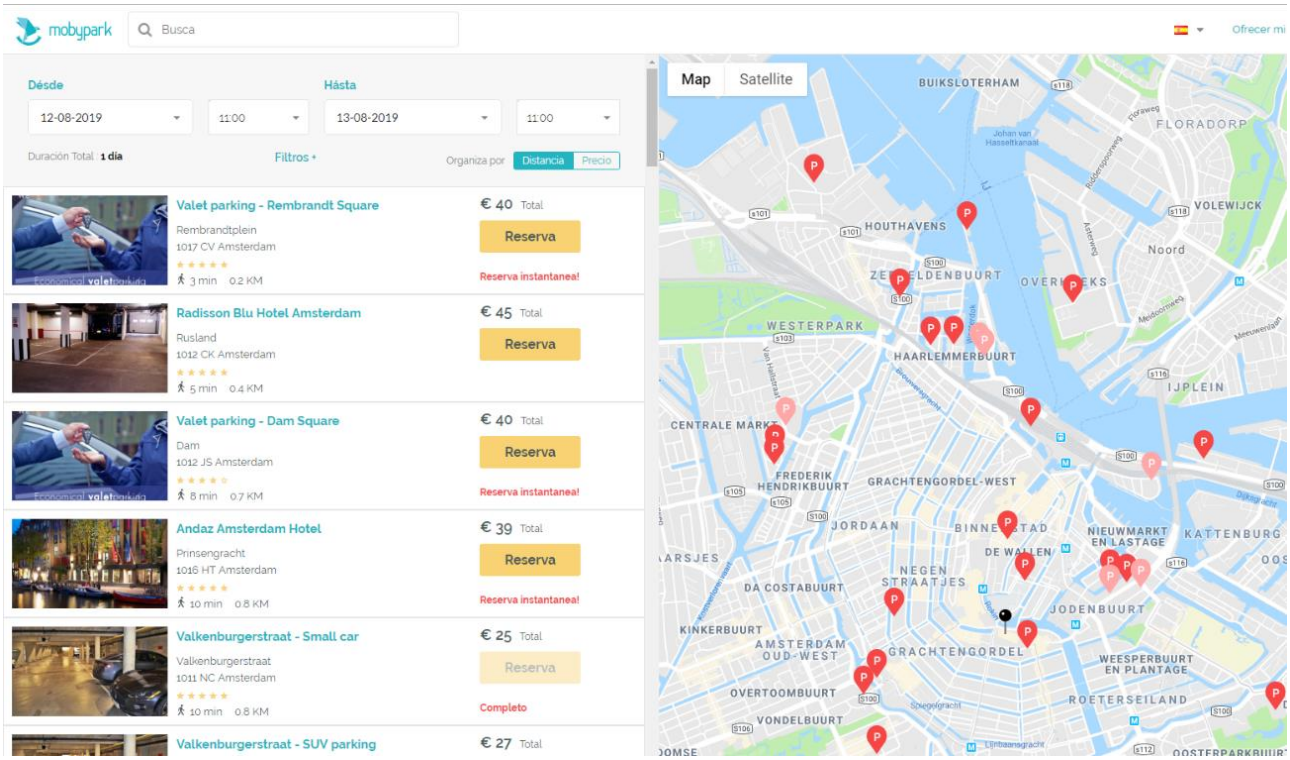


Figura 7: interfície web de Mobypark

Una de les avantatges (segons l'empresa) que té el fet de llogar una plaça d'algú altre és que el preu es redueix.

També es redueix el temps d'aparcament ja que al tenir una plaça assignada l'usuari no ha de malgastar temps (i diners) buscant-ne una de lliure per la ciutat, sinó que es dirigeix a la plaça que ha llogat directament.

3.5 Resum d'idees i conclusions

Com s'ha pogut veure en aquest capítol, actualment ja hi han serveis d'aparcament intel·ligent que permeten al client reservar la seva plaça i monitoritzar les places i/o pàrquings disponibles en una ciutat, fent així un aparcament més eficient i eficaç.

No obstant, després d'investigar a fons, no s'ha trobat cap pàrquing que:

- Estigui basat o utilitzi una arquitectura F2C.
- Permeti donar una utilitat als vehicles que hi han aparcats, emprant els recursos computacionals que cada vehicle disposa per a l'execució de programes de manera distribuïda (o no).
- Tingui en compte una topologia i una estructura de ciutat intel·ligent com la proposada en aquest projecte (amb agents i jeràrquica).

És per això que és convenient dissenyar un sistema d'aparcament intel·ligent que ens permeti, a més d'un aparcament eficient, tenir la possibilitat d'utilitzar els recursos dels vehicles que hi han estacionats i que es beneficiï dels avantatges del F2C.

CAPÍTOL 4 - Arquitectura F2C

Els següents apartats pretenen introduir l'arquitectura de xarxa que s'ha implementat en el projecte, d'una manera teòrica. En capítols següents s'entrarà més en detall en de quina manera s'ha implementat. També s'explica què ens aporta aquesta arquitectura al projecte i per a que el necessitem.

4.1 Cloud

Tradicionalment, en un model de Smart City, els milions de dispositius IoT que recolecten dades les envien al cloud, on grans centres de dades (*datacenters*) les processen per obtenir algun tipus de conclusió (figura 8).

Les plataformes *cloud* tradicionals disposen de servidors agrupats en grans centres de dades que posen al servei dels usuaris i desarrolladors, no només la potència, sinó també l'adaptabilitat, flexibilitat i escalabilitat característiques de la computació *cloud* (*cloud computing*) a entorns que, per limitacions técniques, no seria possible oferir.

No obstant, l'ús de centres de dades, normalment allunyats tant geogràficament com en termes de xarxa dels dispositius que fan ús dels serveis provoca des de latències altes en la comunicació fins a sobre càrregues a la xarxa.

És per això que l'ús dels serveis *cloud* no resulta adequat per aquells casos en els que el processat de les dades té uns requisits temporals forts, com podrien ser la detecció d'alarmes en una fàbrica o la gestió de la conducció d'un vehicle autònom.

Figura 8: connectar més i diferents tipus de "coses" directament al cloud no és pràctic

4.2 F2C

En contraposició a la infraestructura *cloud* tradicional comentada a l'apartat anterior, altres paradigmes com el F2C (*Fog To Cloud*) proposen la distribució d'aquesta capacitat de còmput cap als extrems de la xarxa. Per això es pretén incorporar una capa anomenada "fog" com a capa intermitja entre el cloud i els dispositius generadors de dades.

Aquest concepte de *Fog Computing* va ser originalment proposat per *Cisco* [14] com una arquitectura cloud distribuïda i allunyada dels datacenters centralitzats mitjançant un gran nombre de nodes distribuïts per la xarxa. Es coneixen com a nodes fog i poden ser desplegats a qualsevol lloc sempre que disposin de connexió a Internet: en una fàbrica, en vehicles, en edificis, semàfors, fanals, etc…

Aquests nodes, per tant, estan físicament molt més aprop dels dispositius que generen les dades si ho comparem amb els datacenters, permetent així connexions molt més ràpides.

La considerable capacitat de còmput d'aquests nodes els permeten dur a terme el processament de una bona quantitat de dades, estalviant així el temps d'enviament als servidors cloud.

No obstant, aquesta nova arquitectura no es planteja com a substitut de les infraestructures cloud tradicionals, sinó com a una extensió de les mateixes.

La potència de cómput, la versatilitat i les capacitats de compartició de recursos que ofereix un centre de dades cloud seguirà sent imprescindible per a moltes aplicacions.

L'estructura del F2C és jeràrquica, on la capa cloud està per sobre de la capa fog i aquesta estaria entre la capa cloud i els dispositius que generen les dades, com per exemple els IoT (capa edge).

Ens la podem imaginar en forma de piràmide tal i com es mostra a la figura 9:

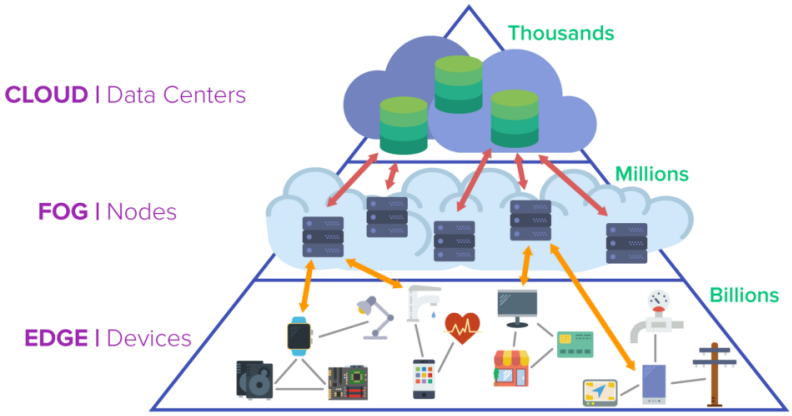


Figura 9: estructura per capes del paradigma F2C

A continuació tenim una taula resum comparativa entre la capa cloud i la capa fog:

	Cloud	**Fog**
Arquitectura	Centralitzada	Distribuïda
Nombre de nodes	Baix	Alt
Latència	Alta	Baixa
Seguretat	Baixa	Alta
Capacitat de còmput	Molt alta	Baixa
Processament de les dades	Lluny d'on es generen	A prop d'on es generen

Taula 3: taula comparativa cloud vs fog

4.3 mF2C

El mF2C [15] és un projecte europeu dedicat a la investigació i al desenvolupament de un nou entorn de treball (*framework*) d'una arquitectura distribuïda que acosti el cloud als usuaris finals.

Per això es vol incorporar la capa comentada a l'apartat anterior, anomenada capa fog, per tal de millorar el rendiment de l'arquitectura de xarxa i beneficiar-se dels avantatges del F2C, comentats a l'apartat anterior.

L'objectiu principal del mF2C és doncs dissenyar i desenvolupar una estructura descentralitzada, oberta, segura i de diversos grups d'interès. Això significa que no té un únic desarrollador sinó que hi col·laboren diverses entitats.

A més, s'espera que el marc de gestió proposat estableixi les bases d'una nova arquitectura de sistemes distribuïts, desenvolupant un sistema i una plataforma de proves on es provaran i validaran casos d'ús semblants als del món real.

A la figura 10 podem observar la pila de recursos del projecte mF2C. Veiem com tenim diferents tipus de fog i cloud. També podem veure que els sensors i actuadors es situen a la part més baixa, mentres que els nodes fog amb més capacitat es situen més amunt.

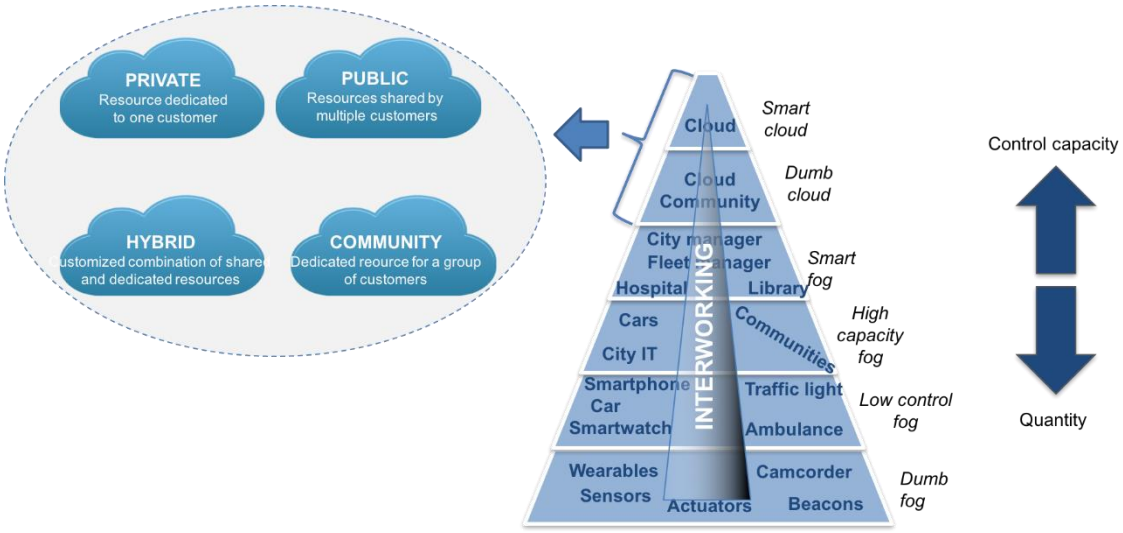


Figura 10: pila de recursos mF2C

4.4 Fog to Cloud en el projecte

Com s'ha vist en aquest capítol, existeixen serveis dins d'una ciutat intel·ligent que necessiten d'una baixa latència per a funcionar correctament. Un dels grans tipus d'aquests serveis són els serveis en temps real.

Com veurem més endavant en aquesta memòria, podem tenir serveis d'aparcament intel·ligent en temps real que necessiten d'una ràpida resposta per part del sistema (ja que el client vol aparcar en un moment donat i no vol esperar).

Pel que fa la part de computació, hem vist que els sistemes cloud tradicionals ofereixen una capacitat de còmput molt més elevada però a un cost elevat de comunicacions i latència.

Dit això, el F2C ens serveix per:

1. L' execució de aplicacions / serveis / programes que no necessiten una gran capacitat de còmput però sí un *delay* / latència baixa.
2. Oferir serveis d'aparcament intel·ligent en temps real.

3. Oferir serveis de conducció autònoma. Aquest punt no pertany al projecte en si, però sí a un dels projectes amb el qual hem integrat el nostre servei d'aparcament i per tant ens podem beneficiar ja que el servei és nostre.
   Recordem que la conducció autònoma requereix latències molt baixes perquè el vehicle respongui ràpidament a les peticions que se li fan.

CAPÍTOL 5 – ANÀLISI DE REQUISITS

Abans de començar a dissenyar i implementar els objectius del projecte, necessitem saber quines funcionalitats i requisits haurà de tenir la nostra plataforma.

En aquest capítol es tracten justament aquests requisits per tal de deixar ben definit fins a on s'arribarà al final del projecte. Recordem que aquest tipus de projectes són "infinits" en el sentit de que sempre podriem estar afegint més funcionalitats i millores i per tant no acabaria mai.

5.1 Contextualització del projecte

Abans de definir els requeriments i les funcionalitats de cada element del projecte cal conèixer una sèrie de conceptes importants per a poder entendre'l.

5.1.1 CRAAX

El CRAAX (Centre de Recerca d'Arquitectures Avançades de Xarxa) és un grup d'investigació multidisciplinari dedicat a dur a terme investigacions avançades en l'àmbit de les xarxes de computadors per tal de oferir els resultats de les investigacions al sector industrial, però també per obtenir solucions que tinguin un gran impacte en la societat en general, al mateix temps que formen professionals altament qualificats.

Està format per professors del Departament d'Arquitectura de Computadors de la UPC (Universitat Politècnica de Catalunya), estudiants de màster o doctorat (PhD) relacionat amb enginyeries de computació i telecomunicació i també per membres del departament d'innovació del Hospital Clínic de Barcelona [16].

Com ja s'ha comentat al capítol 2, el CRAAX actua com a client al qual se li entregarà el projecte un cop finalitzat.

5.1.2 Agents

Un agent és un dispositiu que, al integrar-se / instal·lar-se / col·locar-se dota de certa intel·ligència i capacitat de còmput a un element quotidià de la ciutat. Aquest element pot ser un vehicle, un edifici, semàfors, fanals, entre d'altres. **En el projecte tindrem en compte que cada vehicle té un agent integrat.**

Els agents són capaços de comunicar-se entre sí, i ens permeten executar serveis dins de la smart city.

Per posar un exemple, podriem convertir una Raspberry Pi [17] en un agent instal·lant-hi el software necessari i posteriorment integrar-la a un vehicle.

En el projecte només s'han implementat a nivell de software (i no de hardware), per tant no tenim cap dispositiu com a tal sinó que els tenim en forma de procés corrent dins d'un ordinador.

El conjunt de tots els agents d'una ciutat en formen la seva topologia. Aquesta topologia no pot tenir qualsevol forma, sinó que estarà basada en una estructura jeràrquica semblant a la del F2C.

Dita topologia s'haurà d'emmagatzemar en una base de dades per tal de tenir-la controlada en tot moment.

Cada agent pot tenir una funció diferent segons el seu posicionament en l'estructura jeràrquica i les seves funcionalitats:

Agent: situat a la part més baixa de l'estructura, en una ciutat real n'hi hauria milers o inclús milions. Aquest dispositiu aniria incrustat a cada element intel·ligent de la ciutat, com per exemple als vehicles. Cada agent només està connectat a un agent leader, que correspondrà a l'agent leader de la zona de la

ciutat on es trobi situat. Si l'agent canvia de zona, deixarà d'estar connectat al leader de la zona que ha abandonat i es connectarà al leader de la zona que ha entrat. **No mantindrem connexions entre agents, només en el mòdul RT (RunTime), explicat més endavant.**

Agent Leader: localitzat a la part mitja de l'estructura, per sobre dels agents. Tindrem un leader per zona. Es capaç de comunicar-se amb els agents de la seva zona (no amb tots els de la ciutat) i amb l'agent cloud. S'encarrega de gestionar les peticions que li arriben dels agents de la seva zona o bé d'agents d'altres zones, si el leader de la zona en qüestió no pot resoldre la petició.

Agent Cloud: localitzat a la capa cloud (part més alta de l'estructura), **només n'hi ha 1**. Al estar situat al punt més alt de l'estructura jeràrquica té una visió de tot el sistema. És capaç de comunicar-se amb la base de dades topològica i amb tots els agents leaders de la ciutat. La seva principal funció serà comunicar els diversos agents leaders.

Cal notar que només tenim una definició i un concepte d'agent (i no 3), però aquest pot tenir una de les 3 diferents funcions esmentades.

5.1.3 Testbed

El Testbed és una plataforma situada al laboratori del CRAAX que ens serveix per a dur a terme proves que per limitacions evidents no podem fer en una ciutat real. Té unes dimensions d'aproximadament uns 15 metres quadrats i consta de carreteres, edificis, vehicles, fanals i semàfors que interactuen entre ells. A la figura 11 veiem una fotografia que mostra una vista general del testbed:

Figura 11: fotografia general del testbed

Disposa d'una zona d'aparcament amb quatre places, tot i que només una es troba operativa. En la plaça operativa hi trobem un lector de RFID (Radio Frequency Identification, la tarjeta blanca visible a la figura 12), que ens permet identificar la plaça i detectar si un vehicle està sobre ella. Dita zona d'aparcament la podem observar a la figura 12.

En aquest projecte s'utilitza per a realitzar el servei d'aparcament, explicat a l'apartat 9.2 d'aquesta memòria.

Figura 12: fotografia de la zona d'aparcament del testbed

5.2 Requisits funcionals dels agents

Els agents han de poguer comunicar-se entre sí. Això significa que han de poguer enviar-se missatges tals com peticions de serveis, pas de fitxers i també connectar-se a les bases de dades (en apartats següents veurem que són).

A més, tenim un requisit important imposat per part dels clients: cada agent ha d'estar format per **mòduls independents**, cadascun encarregat de tasques diferents.

La principal funcionalitat dels agents serà la gestionar les peticions de serveis fetes per un client i també ser capaços d'executar-los correctament.

5.3 Requisits funcionals del panell d'administració (front-end)

Com ja s'ha comentat amb anterioritat, necessitarem d'una aplicació web destinada a l'administrador del pàrquing. Per al desenvolupament d'aquesta aplicació ens centrarem en la seva funcionalitat i en la seva usabilitat.

La funcionalitat la podem definir com alguna cosa que es caracteritza per tenir alguna utilitat eminentment pràctica. Haurà de ser, doncs, un front-end plenament funcional, més enllà de qüestions estètiques o eficients.

La usabilitat d'una pàgina web o programa informàtic és la qualitat que presenta per ser sencilla d'utilitzar. Facilita la lectura dels textos o imatges i presenta funcions i menús sencills, per el que l'usuari satisfà les seves consultes i li resulta còmode d'utilitzar.

A continuació es defineixen els mòduls i les funcionalitats que el front-end tindrà. Podem definir cada mòdul de l'aplicació com una part de l'aplicació amb unes funcionalitats concretes. Aquestes funcionalitats han sigut acordades amb el client.

5.3.1 Interfície d'usuari

Requerim d'una interfície d'usuari simple i intuitiva on l'administrador del pàrquing hi pugui veure en tot moment el seu estat i accedir fàcilment a les següents funcionalitats (mòduls):

- Computació: enviar a computar un fitxer al clúster del pàrquing, així com també veure l'estat dels fitxers que s'estan computant en temps real i el resultat/sortida dels fitxers ja computats.

- Llistat de vehicles del pàrquing: taula on hi figura cada vehicle estacionat i la seva informació rellevant.
- Vista general del pàrquing: esquema del pàrquing que mostra gràficament les seves places. També permet aplicar filtratges per veure gràficament l'ocupació del pàrquing o els vehicles que formen part del clúster de computació.
- Dashboards: múltiples gràfiques que ens mostren informació i estadístiques del pàrquing, en temps real.
- Testbed: esquema del pàrquing del testbed que mostra gràficament les seves places, així com també informació dels vehicles que hi ha a cada plaça.

5.3.2 Mòdul d'inici

Aquest mòdul es pot veure com a pantalla d'inici on l'administrador faria el log-in (inici de sessió) amb la seva contrasenya. En aquest projecte el log-in s'ha obviat ja que no ens aporta cap benefici (ningú més que nosaltres entrarà al panell).

Funcionalitats específiques:

- Ha de permetre escollir quantes plantes tindrà el pàrquing.
- Ha de permetre escollir quantes places tindrà cada planta del pàrquing.

Atès que podem tenir múltiples pàrquings en una mateixa ciutat, **necessitem que al arrancar el front-end puguem configurar-lo segons les dimensions de cada pàrquing**.

5.3.3 Mòdul de computació

Aquest mòdul és el que ens permetrà enviar a computar fitxers que tinguem a la nostra màquina (en local). Aquesta mateixa pantalla és la que tindrien els nostres clients que vulguin utilitzar el servei de computació. Cal destacar que només s'admetran fitxers amb extensió .py (fitxers Python, versió 2 o 3).

Funcionalitats específiques:

- Ha de permetre enviar un fitxer localitzat a la màquina local des d'on s'està executant l'aplicació front-end a computar. Aquests fitxers podran ser Python2 o Python3, i podran utilitzar ParallelPython o no. En cas afirmatiu, el programa s'executarà de manera distribuïda, sinó s'executarà de forma seqüencial en una sola màquina.
- Ha de tenir un llistat en forma de taula on puguem visualitzar els fitxers que s'han computat en el passat o que s'estan computant. A més, aquesta taula disposarà d'altra informació d'interès (per exemple, el temps que ha trigat l'execució de cada programa).
- Cada fila de la taula esmentada ha de tenir una columna "resultat" que al fer clic sobre el valor de cada fila en aquesta columna ens permeti descarregar el fitxer que conté la sortida (el resultat) del programa que s'ha executat.
- Ha de permetre esborrar els fitxers de la taula. Un cop ens haguem descarregat el resultat de l'execució, podrem esborrar la fila del fitxer en qüestió de la taula.

5.3.4 Mòdul llistat de vehicles

Funcionalitats específiques:

- Ha de mostrar un llistat en forma de taula on hi figurin tots els vehicles estacionats al pàrquing.
- La taula ha de mostrar informació rellevant de cada vehicle. En concret:
    - Matrícula
    - Si forma part del clúster de computació o no.
    - Si el client ha pagat o encara no.
    - Recursos disponibles (memòria RAM i nº de CPU's).
    - Data d'entrada al pàrquing.

- Ha de contenir un filtre per tal de que mostri només els vehicles els quals la seva matrícula coincideixi amb la matricula entrada per teclat. Aquest filtratge es farà de forma instantània conforme es van entrant els caràcters, és a dir, l'usuari no haurà de prémer cap tecla ni botó per a que s'apliqui el filtre.

5.3.5 Mòdul vista general

Funcionalitats específiques:

- Ha de representar gràficament el pàrquing, amb un esquema inventat (ja que no tenim un pàrquing real).
- Ha de permetre filtrar per planta, és a dir, poder visualitzar cada planta del pàrquing amb la informació dels vehicles estacionats en ella.
- Ha de comptar amb un filtre per:
    - Mostrar gràficament la ocupació del pàrquing.
    - Mostrar gràficament els vehicles que formen part del clúster de computació.

5.3.6 Mòdul dashboards

Funcionalitats específiques:

- Ha de mostrar un gràfic amb el nombre de reserves que hi ha en aquest moment. Com que el servei d'aparcament es farà al testbed, aquest gràfic es referirà a les reserves fetes per al pàrquing del testbed.
- Ha de mostrar un gràfic on puguem visualitzar-hi estadístiques del pàrquing en temps real (% d'ocupació, % de vehicles que computen o % de places reservades).
- Ha de mostrar un gràfic on puguem visualitzar-hi el nº de fitxers computats al pàrquing en els darrers 7 dies.
- Ha de mostrar un gràfic on puguem visualitzar-hi els recursos disponibles del pàrquing en temps real (memòria RAM i nº de CPU's totals del clúster de computació).
- Ha de mostrar un gràfic on puguem visualitzar-hi el nom dels fitxers que s'estan executant al pàrquing, en temps real.

5.3.7 Mòdul Testbed

Aquest mòdul és molt semblant a la vista general, però en aquest cas l'esquema del pàrquing no és inventat sinó que correspon a la zona d'aparcament del testbed.

Funcionalitats específiques:

- Ha de representar gràficament el pàrquing del testbed (només la zona d'aparcament i cap altre element del testbed).
- Ha de permetre clicar a cadascuna de les places. Al fer clic, si la plaça està ocupada, mostra informació d'interès del vehicle estacionat en aquella plaça.
- Cada plaça del pàrquing estarà pintada de color vermell si està ocupada o de color verd si està lliure.

5.4 Aplicació client

Necessitarem algun tipus d'aplicació per a que els nostres clients puguin sol·licitar els serveis que s'implementaran en el projecte.

Els requisits funcionals per aquesta aplicació seran:

- Ha de permetre sol·licitar el servei d'aparcament intel·ligent.
- Ha de permetre sol·licitar el servei de supercomputació o computació.

Serà una aplicació molt concreta i minimalista **destinada únicament al fet de poder sol·licitar serveis**.

CAPÍTOL 6 – DISSENY DEL SISTEMA

L'objectiu d'aquest capítol és determinar quina serà l'arquitectura del nostre model F2C i el disseny de cada component de l'estructura i altres característiques, tals com les comunicacions que s'estableixen entre els components o quins són els missatges que s'envien i de quina manera.

A més, s'explica la tecnologia triada per al panell d'administració front-end i per a l'aplicació pels clients i com s'integraran en l'estructura F2C.

6.1 Estructura principal de la ciutat intel·ligent

Disposem d'una arquitectura jeràrquica basada en F2C. Podem definir dues capes: la capa cloud i la capa fog.

Amb els agents definits al capítol anterior (recordem que són l'element base de la nostra topologia) definirem una estructura jeràrquica com la mostrada a la figura 13:

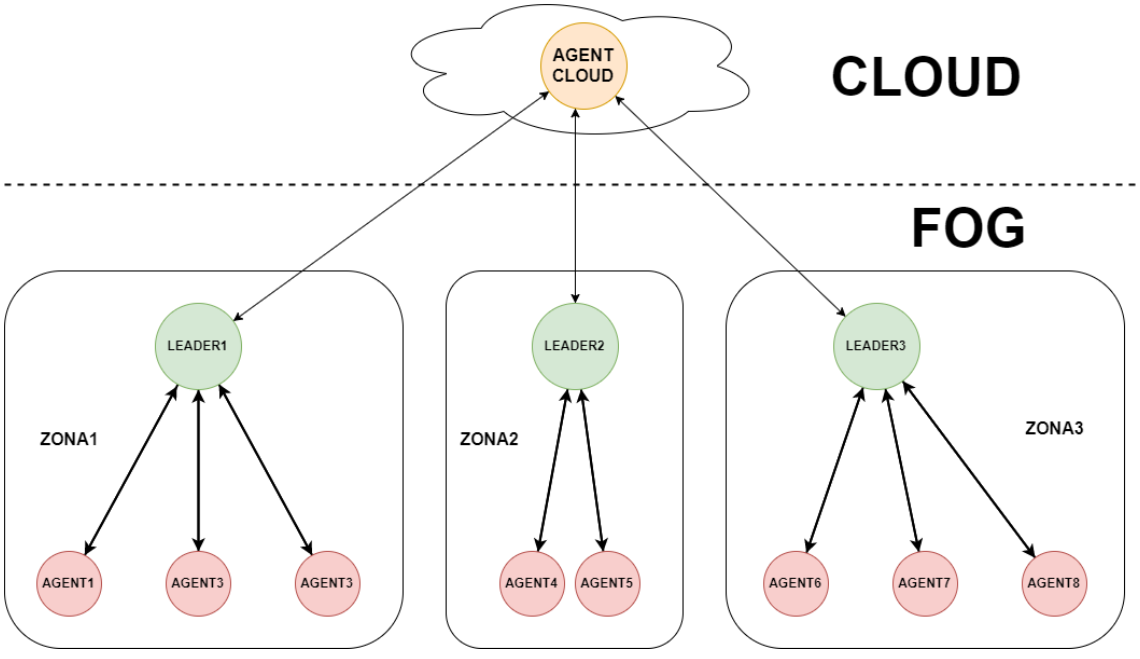


Figura 13: estructura principal de la ciutat a nivell d'agents i F2C

Apreciem com cada zona de la ciutat té un agent leader i diversos agents que estan connectats a ell. Podem definir una zona com una subàrea de la ciutat que es composa per un agent leader i diversos agents. Cada zona és independent de les altres, però totes estan comunicades per el cloud ja que l'agent cloud comunica els respectius leaders.

Llavors, cada vegada que un agent abandona una zona i entra en una altra, deixa de comunicar-se amb l'agent leader de la zona que ha abandonat i es comunica amb l'agent leader de la nova zona.

A més, es disposa d'una base de dades topològica per a guardar la informació de cadascun dels agents de la ciutat, explicada més endavant.

6.1.1 Estructura principal d'un pàrquing F2C

Un pàrquing de la ciutat intel·ligent té la mateixa forma que les zones que hem definit a l'apartat anterior a la figura 16. Així, cada pàrquing estarà format per un agent leader i un conjunt d'agents connectats a ell. Aquest conjunt d'agents està format per cadascun dels agents de cada vehicle estacionat al pàrquing.

No obstant, hi ha una diferència: el leader del pàrquing ha d'estar connectat a la base de dades del pàrquing del qual n'és leader, ja que haurà obtenir i manipular la seva informació. Cada pàrquing de la ciutat té una base de dades pròpia.

La següent figura mostra l'estructura del pàrquing a nivell d'agents i F2C:

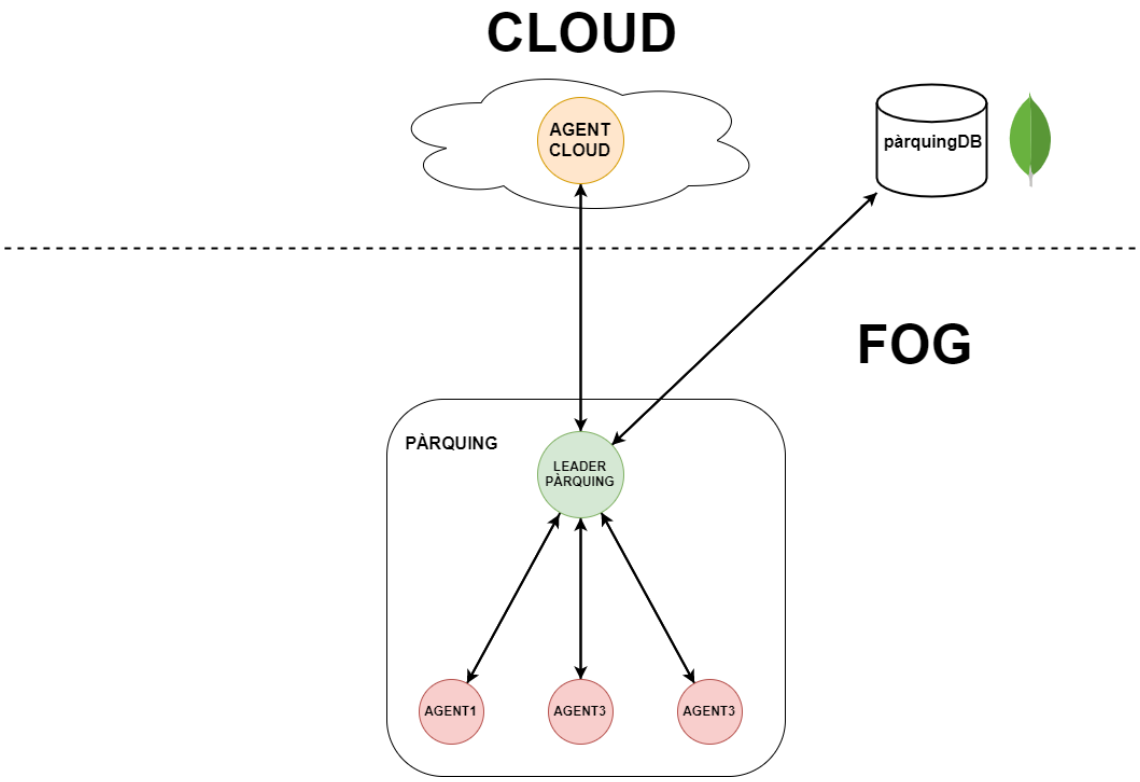


Figura 14: estructura d'un pàrquing F2C a nivell d'agents

6.2 MongoDB

MongoDB [18] és un programari de codi obert per a la creació i gestió de bases de dades on hi guardem les dades en format JSON (parells clau-valor, format també conegut com a diccionari). Una base de dades de MongoDB està formada per una o més bases de dades que els hi diem col·lecció. Les col·leccions estan formades per documents, que són dades en format JSON.

Es tracta d'un sistema de base de dades no estructurada (també anomenat no SQL). El fet de que sigui no estructurada significa que cada element pot tenir camps diferents dins la base de dades.

Aquest tipus de base de dades proporcionen una major escalabilitat que les estructurades o SQL (MySQL [19] per exemple) i per tant són adequades per aquest tipus de projectes ja que en el nostre cas, per exemple, podriem arribar a tenir milions d'agents en una mateixa ciutat.

JSON (JavaScript Object Notation) és un format de text sencill per a l'intercanvi de dades. Un dels avantatges d'aquest format és que és independent del llenguatge de programació que utilitzem, i la majoria de llenguatges l'accepten.

Aquest format, també anomenat diccionari, guarda la informació en parells clau – valor tal i com es pot veure a la figura 15:


```
_id: "21"
host: "192.168.1.38"
port: 9000
device: "CloudAgent"
role: "cloud"
IOT: "IOT"
leaderIP: "192.168.1.38"
status: 1
nodeID: "0"
```


Figura 15: Exemple d'agent a la base de dades topològica

6.2.1 Base de dades topològica

Com ja s'ha comentat a l'anterior apartat, comptem amb una base de dades MongoDB per guardar la topologia de la ciutat intel·ligent. Es tracta d'una base de dades centralitzada (només n'hi ha una) situada en un servidor al cloud.

En aquesta base de dades s'enregistren o s'esborren els agents cada vegada entren o surten de la ciutat, es modifica la seva informació quan canvien de zona/leader, etc. Notem que en aquesta base de dades topològica només hi guardem la informació de tots els agents de la ciutat (i no cap altre dispositiu com ara els IoT).

Es va acordar amb el client quina seria la informació de cada agent que guardariem:

- Adreça IP
- Port
- Adreça IP del seu leader
- Funció (agent, agent leader o agent cloud)

- Tipus de dispositiu (ambulància, edifici, etc…)
- IoT (dispositius IoT associats a l'agent)
- Status (1 actiu, 0 inactiu)
- ID (identificador únic per diferenciar-lo de la resta)

L'altre col·lecció que trobem en aquesta base de dades és el catàleg de serveis. Aquest conté la informació de cada servei ofert a la smart city.

Cada servei tindrà els següents camps dins de la base de dades (figura 16):

- ID del servei: identificador del servei dins del catàleg per diferenciar-lo dels altres.
- Codi: fitxer que conté el codi que s'executarà quan es demani el servei.
- Descripció: Breu descripció sobre el servei.
- Arguments: paràmetres d'entrada que el servei necessita.
- IoT: dispositius IoT i agents que intervenen en l'execució del servei.
- Output: resultat que produeix l'execució del servei.

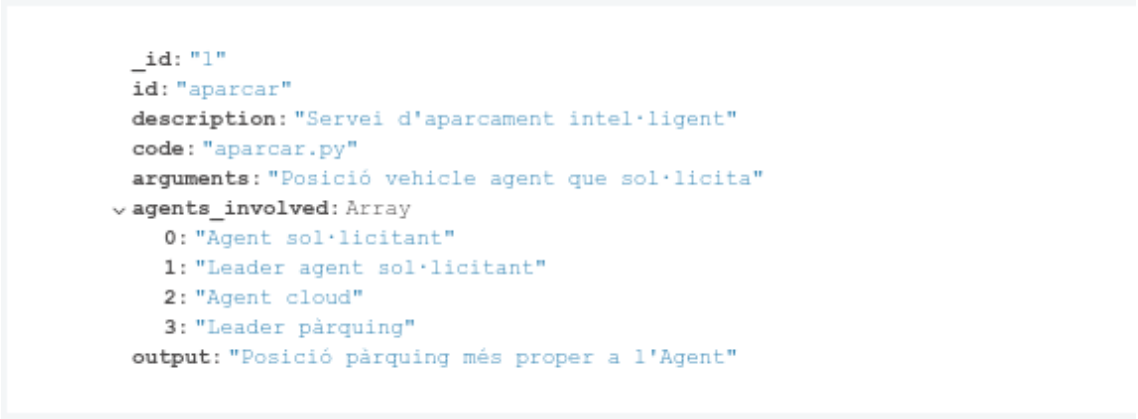


Figura 16: exemple de servei al catàleg de serveis

6.2.2 Base de dades d'un pàrquing F2C

Cada pàrquing de la ciutat té una base de dades pròpia per guardar-hi la informació de les seves places, plantes, vehicles que es troben dins del pàrquing, reserves de plaça fetes, entre d'altres.

Més concretament, en aquesta base de dades hi guardarem:

- Les reserves fetes per al pàrquing.

- Històric de fitxers computats.
- Fitxers que actualment s'estan computant.
- Nombre de places.
- Nombre de plantes.
- Vehicles estacionats i la seva informació més rellevant.

6.3 Agents

El disseny proposat pels agents va ser definit pels clients. Per a guardar la informació general d'un agent, es va acordar fer-ho en format JSON, així com també els camps que ens interessa guardar per a cada agent:

```
agent_info = {
    "host" : "IP de l'agent",
    "port" : "port on escolta la API",
    "leaderIP" : "IP del seu leader",
    "IoT" : "dispositius IoT,
    "role" : "funció de l'agent",
    "device" : "tipus de dispositiu"
}
```

A més de la seva informació, un agent està composat per els següents mòduls tal i com podem observar a la figura 17:

- API
- TRM (Topology Resources Management)
- SEX (Service Execution)
- RT (Run Time)

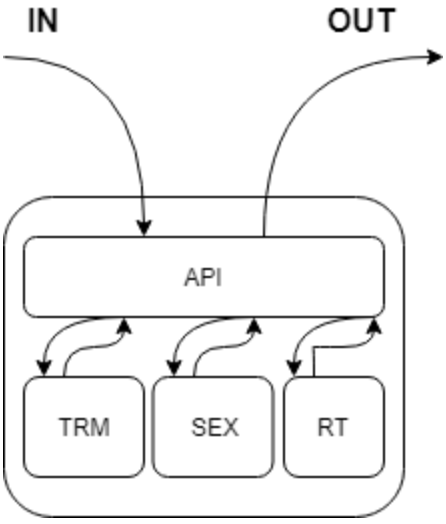


Figura 17: disseny modular dels agents

Aquest disseny modular facilita els canvis posteriors i generalitza l'agent per tal de que pugui executar qualsevol servei (sempre hi quan estigui definit al SEX i al catàleg de serveis).

Els apartats que segueixen expliquen en detall cada mòdul de l'agent.

6.3.1 API

Utilitzarem el model REST per a la API. Aquest model utilitza peticions HTTP per obtenir, escriure, modificar o esborrar dades.

Les peticions HTTP poden ser dels següents tipus, en funció de què es vol fer amb les dades:

Petició GET: Petició utilitzada per a obtenir dades o recursos.

Petició POST: Petició utilitzada per a escriure dades. Normalment cal enviar un body (cos) amb les dades

Petició PUT: Petició usada per a modificar dades. Cal enviar un body amb els camps que volem modificar.

Petició DELETE: Petició que ens serveix per esborrar dades. Cal enviar un body amb un identificador per a les dades que volem esborrar.

La API ens serveix per a la entrada i sortida de dades amb l'exterior. És a dir, la API ens servirà per a que l'agent es pugui comunicar amb altres agents o elements de la ciutat intel·ligent.

6.3.2 Mòdul TRM

El TRM (Topology Resources Management) s'encarrega de les operacions d'inserció, modificacions o eliminacions de la base de dades topològica.

Així, cada vegada que a l'agent li arribi una petició d'aquest tipus, la API cridarà a la funció pertinent del TRM encarregada de fer la operació corresponent.

6.3.3 Mòdul SEX

El mòdul SEX (Service Execution) s'encarrega de rebre i gestionar les peticions de serveis que se li fan a un agent. Més concretament, quan li arriba una petició de servei s'encarrega d'anar a buscar la informació del servei sol·licitat al catàleg de serveis i posteriorment el codi del servei en qüestió al servidor SFTP.

Un cop té aquesta informació, li envia la petició d'execució del servei a la API, i aquesta la redirigeix cap al mòdul RT.

6.3.4 Mòdul RT

Mòdul RT (Run Time): Correspon a l'execució de les peticions de servei gestionades pel SEX. S'encarregarà doncs d'executar el codi pertinent de cada servei.

6.4 Panell d'administració

El panell d'administració front-end és una interfície web accessible des de qualsevol navegador de quasi qualsevol dispositiu que disposi de connexió a Internet (veure restriccions a l'apartat 5.4.1).

L'arquitectura d'aquest front-end es basa en el conegut **model client-servidor**, en el qual un o més clients fan peticions a un servidor, aquest les gestiona i els hi serveix la informació demanada.

A continuació es detallen les tecnologies que s'han decidit emprar per a cadascuna de les parts client i servidor.

6.4.1 Servidor web

A la part del servidor utilitzarem Node.js [20]. Node.js és un entorn de servidor de codi obert. Corre sobre un motor Javascript anomenat V8 i desenvolupat per la companyia *Google* [21]. Node.js pot executar-se en en diverses plataformes (Linux, Windows, Mac OS X, etc..) i utilitza el llenguatge de programació Javascript per als seus fitxers de servidor.

Es va decidir utilitzar aquest entorn ja que és un dels més coneguts i utilitzats arreu del món per a programació d'aplicacions web que necessitin d'un servidor (això significa que a Internet hi ha molta informació i manuals d'ajuda) i també perquè s'adapta als nostres requisits.

Algunes de les seves característiques més importants són:

1. Escalabilitat
2. Asincronia
3. Fàcil d'apendre
4. Molt popular i usat entre els desenvolupadors d'aplicacions web
5. Utilització del llenguatge Javascript a la part del servidor

6.4.2 Framework Express

Express [22] és un framework web de NodeJS que ens ajuda a organitzar la nostra aplicació web en una arquitectura MVC (Model Vista Controlador) en la part del servidor. Es tracta del framework de Node.JS més utilitzat i ens ajuda a manejar les rutes, les peticions i les vistes de l'aplicació.

El desenvolupament d'aquesta plataforma s'ha elaborat segons l'esquema de la figura 18:

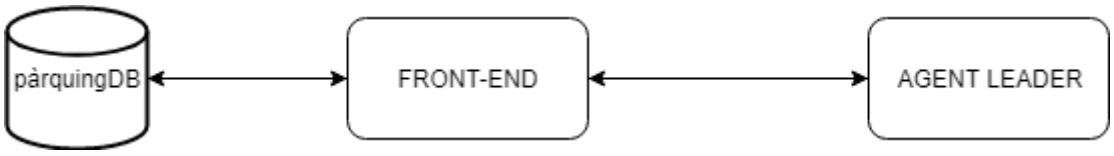


Figura 18: connexions del panell d'administració

El front-end està connectat amb la base de dades del pàrquing per a poder obtenir i escriure informació. També està connectat amb l'agent leader del pàrquing. Connectat significa que pot enviar-li peticions a la API.

6.4.3 Client web

El client és l'usuari del front-end que interactua amb l'aplicació, en aquest cas l'encarregat d'administrar el pàrquing.

Per al desenvolupament d'aquesta aplicació s'utilitzaran les tecnologies que avui en dia quasi totes les aplicacions web empren: els llenguatges de programació HTML, CSS i Javascript, entre d'altres.

Tal i com s'ha dissenyat, el panell d'administració consta d'una vista genèrica per a tots els mòduls, formada per tres elements:

- Barra lateral (a la part esquerra de la pantalla) de navegació que ens serveix per anar d'un mòdul a un altre.
- Header o capçalera per indicar el nom del módul on estem i altres eines o botons d'utilitat.
- Body o contingut del mòdul.

Llavors, un primer disseny visual del panell seria el mostrat a la figura 19:

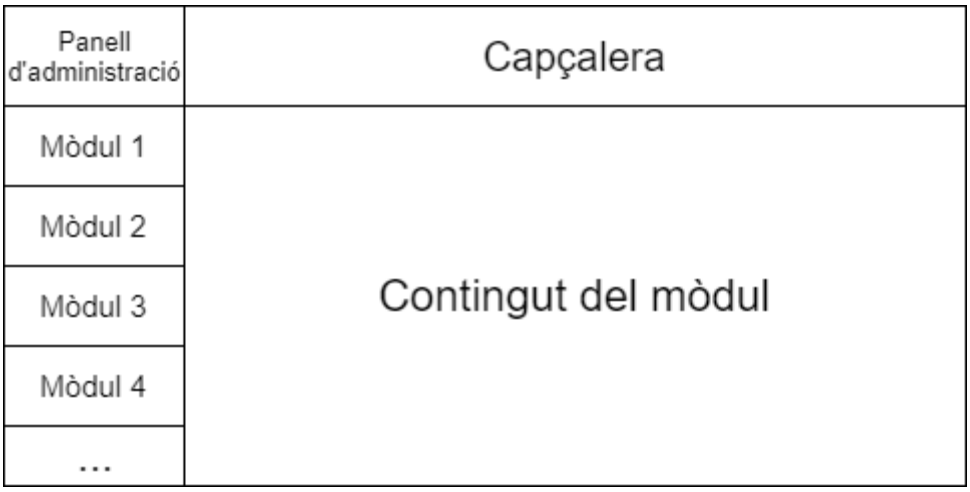


Figura 19: estructura principal del panell d'administració

6.4.4 Estructura de fitxers i MVC

L'arquitectura Model-Vista-Controlador (MVC) és un patró de disseny utilitzat per a la implementació d'interfícies d'usuari. Aquest patró de desenvolupament de programari divideix l'aplicació en tres parts interconnectades: el model de dades, la interfície d'usuari i la lògica de control. És un patró utilitzat freqüentment en aplicacions web, doncs es basa en les idees de reutilització de codi i la separació de conceptes, característiques que busquen facilitar el desenvolupament d'aplicacions i el seu posterior manteniment.

El Model: s'encarrega d'enviar a la Vista aquella part de la informació que en cada moment es demana per a que sigui mostrada a l'usuari.

El Controlador: respon a events (usualment accions de l'usuari) e invoca peticions al Model quan es fa alguna sol·licitud sobre la informació (per exemple editar un document o registre en una base de dades). En molts casos el controlador correspon al sistema gestor de bases de dades (en aquest cas correspondria a MongoDB).

La Vista: presenta el model (informació) en un format adequat per a que l'usuari hi pugui interactuar. La vista normalment correspon a la interfície d'usuari.

La figura 20 mostra el flux que s'estableix en el MVC:

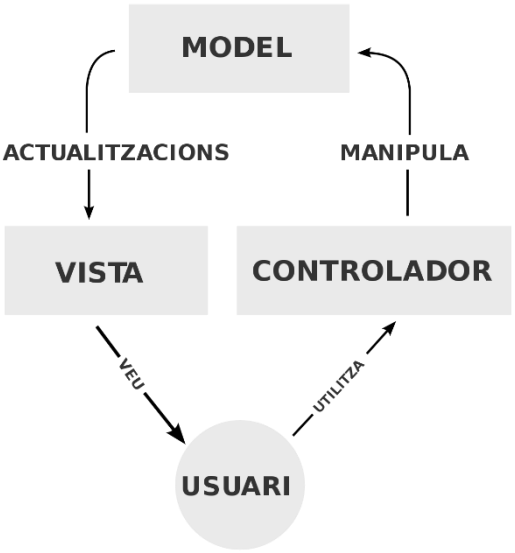


Figura 20: funcionament del patró MVC

L'estructura de fitxers per al panell d'administració és la següent:

- Directori views: conté els fitxers que defineixen les vistes de cada mòdul de l'aplicació.
- Directori node_modules: conté els fitxers corresponents als mòduls de Node.JS.
- Directori routes: conté els fitxers que defineixen les rutes de l'aplicació.
- Directori public: conté fitxers auxiliars per a la implementació de cada mòdul. Podriem dir que són fitxers auxiliars dels fitxers del directori Views
- Fitxer package.json: en ell s'hi troben totes les dependències a instal·lar abans de fer córrer l'aplicació.
- Fitxer server.js: en aquest fitxer es defineix la configuració i els mòduls de Node.JS utilitzats en l'aplicació.

6.5 Aplicació client

Per al desenvolupament d'aquesta aplicació s'ha utilitzat també Node.JS en la part del servidor. Es tracta d'una aplicació web (accessible des d'un navegador web).

S'ha aprofitat l'estructura del panell d'administració, tot i que aquesta aplicació al ser molt més petita (ja que només és per a demanar serveis) no requereix tants fitxers ni mòduls.

Una altre possibilitat era una aplicació mòbil (app), però a causa de la complexitat tècnica es va descartar.

Llavors, l'aplicació tindrà dues pantalles:

1. Pantalla d'inici que permetrà seleccionar el servei a sol·licitar.
2. Pantalla per a sol·licitar el servei de supercomputació o computació.

En el cas del servei d'aparcament no tindrem cap altre pantalla ja que amb un sol clic al botó "APARCAR" ja en tindrem prou.

En canvi, per al servei de computació, l'usuari haurà d'entrar una sèrie de paràmetres tals com el fitxer a computar, la versió de Python que vol utilitzar i si el fitxer enviat utilitza Parallel Python o no.

La figura 21 mostra un esquema de l'aplicació, on els dos rectangles grans corresponen a les dues pantalles:

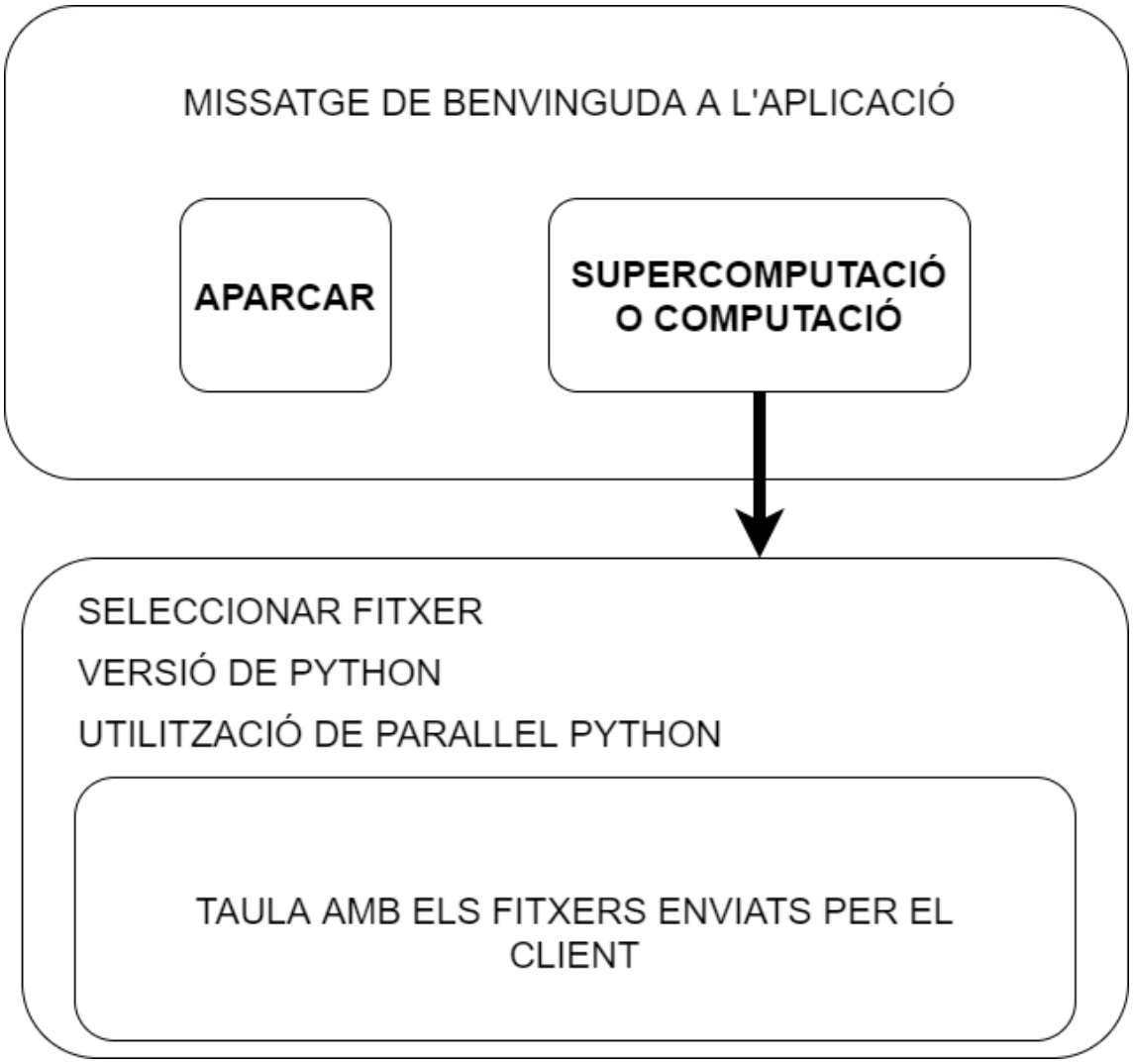


Figura 21: disseny de les dues pantalles que conformen l'aplicació

6.6 Aspectes generals d'un pàrquing F2C

6.6.1 Reserva de places

Quan un vehicle fa la petició del servei d'aparcament, automàticament se li reserva una plaça del pàrquing. No obstant, no se li indica a quina plaça ha d'aparcar fins que arriba al pàrquing.

6.6.2 Assignació de places

Disposem d'un algorisme d'assignació de les places quan un vehicle arriba al pàrquing.

L'objectiu d'aquest algorisme és buscar l'eficiència en l'espai, de manera que les places s'assignaran seqüencialment a mesura que els vehicles arribin al pàrquing. Sempre s'assignarà la plaça més propera (figura 22). No enviarem mai un vehicle a la segona planta fins que no haguem omplert la primera.

Per exemple, no tindria massa sentit enviar un client a la planta nº 3 quan tenim 60 places lliures a la planta nº 0 (planta baixa).

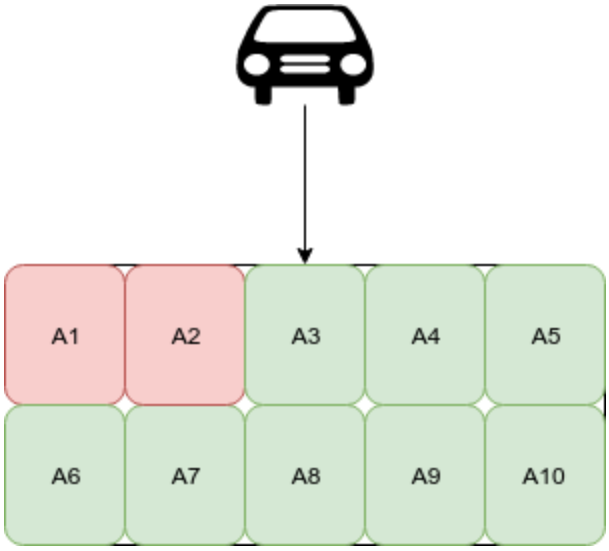


Figura 22: assignació de places seqüencial

Ara, imaginem que el vehicle que ocupava la plaça A2 ha marxat, deixant la plaça lliure. La primera plaça disponible i la que li assignarem al proper vehicle que entri serà la A2 (figura 23).

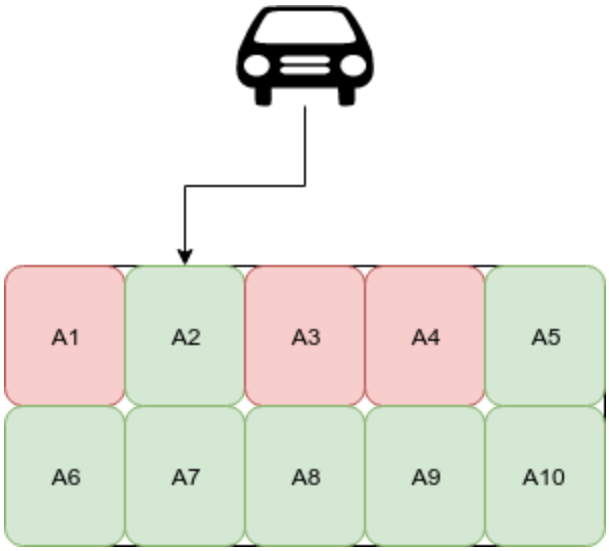


Figura 23: algorisme d'assignació de places

CAPÍTOL 7 – IMPLEMENTACIÓ

El present capítol detalla la implementació de cada element que conforma el projecte.

S'inclouen els troços de codi més rellevants i també captures de pantalla del resultat.

També es mencionen i s'expliquen breument les eines i el programari utilitzat per a dur a terme la implementació del projecte.

7.1 Resum de tecnologies

7.1.2 Agents

- Llenguatge de programació: Python (Versió 3)
- Llibreries de Python utilitzades:
    - pymongo → ens permet connectar-nos i realitzar operacions contra una base de dades MongoDB des de Python.
    - pysftp → ens permet connectar-nos a un servidor SFTP des de Python.

  - requests → ens permet enviar peticions HTTP i rebre'n la resposta des de Python.

7.1.2 Servidor Web

- Llenguatge de programació: Node.JS (Javascript)
- Framework: Express
- Motor de plantilles: Handlebars.js
- Gestor de paquets: npm
- Llibreries de Node.JS utilitzades:
  - Body-parser
  - Multer
  - Cors

7.1.3 Client Web

- Llenguatge de programació: ECMAScript (Javascript)
- Llenguatge de marcatge web: HTML 5 (HTML versió 5)
- Llenguatge d'estils: CSS (Cascading Style Sheets)
- Gràfics web:
  - HTML 5 Canvas
  - Chart.js
- Client HTTP: Axios
- Tecnologia d'asincronia: AJAX
- Frameworks:
  - Bootstrap
  - jQuery

7.1.4 API

- Llenguatge de programació: Python (Versió 3)
- Framework: CherryPy

7.2 Eines per al desenvolupament

7.2.1 Sublime Text

Sublime Text [23] és un editor de text pensat per a programadors, ja que incorpora funcions específiques per a facilitar l'escriptura de codi, a més d'una interfície molt visual i minimalista. Amb aquest editor s'han codificat tant els agents com les aplicacions web.

7.2.2 Postman

Postman [24] és una eina que permet interactuar amb API's RESTful. Aquest client HTTP presenta una interfície amigable i ens permet enviar peticions a la API i rebre les respostes. S'ha utilitzat per a verificar el correcte funcionament de la API dels agents.

7.2.3 VirtualBox

És un software gratuït i de codi obert desenvolupat per Oracle Corporation [25] que permet la creació, configuració i utilització de màquines virtuals [26]. Aquestes màquines virtuals tenen un sistema operatiu propi i es poden configurar per a que tinguin els recursos (CPU i RAM) que nosaltres vulguem.

En aquest projecte s'utilitzarà el sistema operatiu Debian 9.3 [27] (sense entorn gràfic) i cada màquina virtual tindrà 1 CPU i 1 GB de memòria RAM. L'assignació d'aquests recursos no pot ser major que l'assignada ja que sinó el sistema amfitrió (on estem corrent les màquines virtuals) es col·lapsa a causa de la falta de recursos per a poder suportar totes les màquines (entre 10 i 15). La simulació dels agents i les proves de rendiment de Parallel Python del nostre pàrquing s'han fet amb màquines virtuals.

7.2.4 ParallelPython

És una llibreria incorporada en el llenguatge de programació Python que ens permet executar un programa o aplicació (escrit en Python) de forma distribuïda, és a dir, en diverses màquines alhora [28].

Per a que això sigui possible, haurem de distribuïr el nostre programa en tasques atòmiques (és a dir, que no es poden dividir), les quals seran executades entre les màquines que tinguem disponibles.

7.2.5 MongoDB Compass

MongoDB Compass [29] és una GUI (Graphic User Interface o Interfície gràfica d'usuari en català) que ens permet veure el contingut d'una base de dades MongoDB d'una manera gràfica i molt visual, així com crear, modificar i esborrar els seus elements. Principalment s'ha utilitzat per a verificar que els agents, vehicles i altres elements s'hagin emmagatzemat correctament a la base de dades MongoDB.

7.2.6 draw.IO

Aplicació web amb la qual s'han dibuixat els diagrames i esquemes que figuren a la present memòria i també els que s'han fet per a les presentacions (demos) als clients [30].

7.2.7 Secure Shell

Secure Shell o simplement SSH [31] és un protocol de xarxa que ens permet connectar-nos a una màquina remotament, és a dir, des d'una altra. Un cop estem connectats podem executar comandes tal i com fariem des d'una terminal.

SSH incorpora eines/comandes per copiar de forma segura fitxers d'una màquina a una altra (comanda scp), que han sigut de gran utilitat per a copiar fitxers des de la màquina host (màquina amfitrió on estem corrent Docker i VirtualBox) als contenidors de Dockers, així com a les màquines virtuals de VirtualBox.

7.2.8 Docker

Docker és un projecte de codi obert que automatitza la creació i el desplegament d'aplicacions utilitzant contenidors [32]. Els contenidors són instàncies creades a partir d'una imatge, que és la que conté tota la informació relacionada amb l'aplicació que estem automatitzant. Les imatges són reutilitzables, és a dir, a partir d'una imatge podem crear múltiples contenidors.

A diferència de les màquines virtuals, els contenidors de Docker utilitzen els recursos de la màquina host comuns (processador, memòria) per a córrer els contenidors. En canvi, les màquines virtuals són totalment aïllades: tenen els seus propis recursos i sistema operatiu.

Per tant, els contenidors són molt més lleugers que les màquines virtuals. Això fa que en puguem arrencar més a la vegada, sense que el sistema amfritrió es col·lapsi. No obstant, encara que tinguem 200 contenidors corrent, això no farà que tinguem més potència de còmput (ja que els recursos de la màquina host sempre seran els mateixos).

Al projecte s'ha utilitzat per a simular una gran quantitat d'agents.

7.2.9 Navegadors web

Per comprovar que el front-end està actiu i funciona correctament s'ha utilitzat el navegador Google Chrome [33]. També s'ha utilitzat l'eina "Consola" que incorpora el navegador. Podem accedir-hi amb la tecla F12 i ens permet visualitzar missatges d'error o d'alerta (warnings) que es produeixen en la pàgina que estem visualitzant en el navegador. Això ens permet veure a on està fallant el nostre codi i per tant ens facilita la correcció d'errors.

7.3 Implementació de la classe Agent

7.3.1 Definició de la classe

Els atributs de la classe són les característiques que cada objecte de la classe agent té.

A la figura 24 es mostra el troç de codi corresponent als atributs de la classe agent:

```
class agent:

    # Inicialitzem el leader
    def __init__(self, host, port, device, role, leaderIP, IOT):

        # Info del leader per a la topoDB
        self._nodeinfo = {
            'myIP' : host,
            'port' : port,
            'device' : device,
            'role' : role,
            'leaderIP' : host,
            'IOT' : IOT,
            'status' : 1
        }

        # TRM
        self._TRM = trm()

        # SEX
        self._SEX = sex()

        # RT
        self._RT = rt(self)

        # API
        self._api = API(self)
        self._api.start()
```

Figura 24: atributs de la classe agent

Veiem que la classe agent com a atributs el disseny que s'explica a l'apartat 6.3 d'aquesta memòria.
La informació de l'agent ve representada com un diccionari, amb parells de clau valor. El camp "status" pot ser 1 si l'agent està actiu o 0 si està inactiu. Que l'agent estigui actiu significa que està corrent/engegat, mentres que si està inactiu està apagat.

7.3.2 Inicialització d'un agent

El procés d'inicialització d'un agent consta de dos passos:

1. Crear l'objecte agent com a tal.
2. Enregistrar-lo a la base de dades topològica per tal de que pugui executar serveis i formi part de la smart city.

El tros de codi de la figura 25 mostra el pas 1:

```
# Programa principal que s'executa
if __name__ == "__main__":
    # Inicialitza leader
    leader = agent('192.168.1.39', 8000, 'leader_parquing', 'leader', '192.168.1.2', ['IoT 1', 'IoT 2'])
```

Figura 25: creació d'un objecte de la classe agent

On estem creant l'objecte leader corresponent a un agent leader, passant com a paràmetres la informació necessària.

Aquest fragment de codi ha d'anar a un fitxer Python, que és el que executarem per iniciar l'agent.

Per a que s'enregistri a la base de dades topològica, haurem de fer una petició POST a la API de l'agent leader enviant-li els camps necessaris. Per exemple:

POST http://192.168.1.2:8000/register_agent

Per a fer el post utilitzarem la llibreria requests de Python, tal i com mostra la figura 26:

```python
if __name__ == '__main__':
    # IP i port del leader
    leaderIP = '192.168.1.2'
    leaderPort = 8000
    # Info de l'agent
    data = {
        'myIP' : '192.168.139',
        'port' : 8000,
        'device' : 'Agent prova',
        'role' : 'agent',
        'leaderIP' : leaderIP,
        'IoT' : ['IoT 1', 'IoT 2']
    }
    # Petició POST
    req  = requests.post('http://'+leaderIP+':'+leaderPort+'/register_agent', json=data)
```

Figura 26: petició al agent leader per enregistrar un agent a la base de dades topològica

7.3.3 Mòdul TRM

La funció principal d'aquest mòdul és la que ens serveix per afegir agents a la base de dades topològica. El codi d'aquesta funció la trobem a la figura 27:

```python
@cherrypy.expose
@cherrypy.tools.json_in()
@cherrypy.tools.json_out()
# Register agent a la topoDB
def register_agent(self):
    if cherrypy.request.method == "POST":
        body = cherrypy.request.json
        try:
            # Busquem l'ID intern per el nou agent
            nodeID = self.agent_collection.find_and_modify(query={ '_id': 'nodeID' },update={'$inc': {'seq': 1}}, new=True ).get('seq')
            body['_id'] = str(int(nodeID))
            body['status'] = 1
            # Li creem un ID amb la llibreria uuid
            body['nodeID'] = str(uuid.uuid4())
            # L'insertem a la col·lecció
            self.agent_collection.insert_one(body)
        except pymongo.errors.DuplicateKeyError as e:
            nodeID = post_topoDB(body)
        # Li retornem l'ID generat
        return str(int(body['nodeID']))
```

Figura 27: funció per enregistrar un agent a la base de dades topològica

Les funcions per modificar o esborrar un agent són molt semblants. El procediment a seguir és primer trobar l'agent a partir d'un ID i després modificar-lo o esborrar-lo.

Una altra funció interessant (figura 28) d'aquest mòdul és la que utilitzem per obtenir tots els agents connectats a un leader:

```python
@cherrypy.expose
@cherrypy.tools.json_in()
# Obte els agents amb leaderIP = leaderIP
# GET .../get_leader_agents/192.168.1.10
def get_leader_agents(self, ide):
    # Troba tots els agents connectats al leader amb IP donada
    cursor = self.agent_collection.find( { "leaderIP" : ide })
    all_data = list(cursor)
    output = []
    for document in all_data:
        # Només volem els agents que no siguin leader o cloud
        if(str(document['role']) != 'leader' and str(document['role']) != 'cloud'):
            output.append(document)
    agent_json = json.dumps(output, default=json_util.default)
    return agent_json
```

Figura 28: funció per a obtenir els agents connectats a un agent leader amb adreça IP donada

De manera que si fem una petició GET a la API de l'agent leader amb IP 10.0.6.50 i al port 8000 com segueix

GET http://10.0.6.50:8000/get_leader_agents/10.0.6.50

Ens retornarà el llistat d'agents que té connectats i la seva informació.

7.3.4 Mòdul SEX

La classe SEX té com a atribut un array / llista de parells on hi guardem el nom del servei i quin agent l'ha sol·licitat.

Un exemple podria ser:

[('aparcar', agent1) , ('computacio', agent2)]

On l'agent1 ha demanat el servei aparcar i l'agent2 el de computació.

Consta de dos mètodes principals:

1. get_service_request(self, serviceID, agentID) → Gestiona les peticions de servei entrants

Ha de rebre com a paràmetres l'identificador del servei (serviceID) i l'identificador de l'agent que el sol·licita.

El primer que fa aquest mètode és comprovar que l'agent no hagi demanat ja aquest servei. Si no l'ha demanat, afegeix un parell a l'array de serveis (cua de serveis). Si ja l'havia demanat se li retorna un missatge d'error.

El següent pas és anar a buscar la informació del servei al catàleg de serveis.

Quan ja té la informació (i per tant sap quin codi és) es descarrega el codi del servidor SFTP.

Retorna la informació del servei.

La figura 29 mostra el fragment de codi en Python corresponent a la funció:

```
# Rep id del servei a executar i id del agent que demana el servei
def get_service_request(self, serviceID, agentID):
    # Si el mateix agent ja ha demanat el servei rebutjem request
    if((serviceID,agentID) in self._services):
        return 'Ja has demanat aquest servei'

    # sino ens l'afegim a la cua de serveis del SEX
    self._services.append((serviceID,agentID))

    # handle = diccionari amb info del servei
    handle = self.handle_request(serviceID)

    # handle = json amb la info del servei aparcar
    service_code = self.get_code(serviceID)

    # URL del codi del servei descarregat del SFTP
    handle['url'] = '../'+service_code

    return handle
```

Figura 29: funció del mòdul SEX per gestionar les peticions de serveis entrants

2. get_code(self, serviceID) → Descarrega el codi del servei del servidor SFTP

Atès que no tenim servidor SFTP com a tal utilitzarem un directori en local de la màquina per simular-lo. He utilitzat la llibreria pysftp que, donat el path del directori on es troba el fitxer i el path on volem guardar-lo se'ns descarrega (figura 30).

```
# Obte el codi del SFTP del servei donat
def get_code(self, serviceID):
    localPath = '../'+str(serviceID.lower())+'.py'
    remotePath = '/home/aerie/SFTP/'+str(serviceID.lower())+'.py'
    cnopts = pysftp.CnOpts()
    cnopts.hostkeys = None
    s = pysftp.Connection(host='localhost', username="aerie", password="password", cnopts=cnopts)
    s.get(remotePath, localPath)
    return str(serviceID)+'.py'
```

Figura 30: funció del mòdul SEX per descarregar un fitxer del servidor SFTP

7.3.5 Mòdul RT

El mòdul RT consta d'una funció principal per executar els serveis, que és cridada per la API cada vegada que li arriba una petició de servei.

Aquesta funció rep com a paràmetres l'identificador de l'agent que ha demanat el servei, la informació del servei en qüestió i un camp amb informació extra per si fan falta per a executar el servei.

Es fa ús de la llibreria *subprocess* per executar el fitxer que conté el codi del servei. Al executar el servei li passem l'identificador de l'agent i la informació necessària per a cada servei. Com que cada servei necessita una informació diferent per a ser executat, els hem d'executar per separat tal i com mostra la figura 31:

```
# Rep l'ID de l'agent sol·licitant del servei
# i la informació del servei a executar
def executa_servei(self, agentID, info_servei, extraParams):
    # URL o path complet del fitxer a executar
    codi_servei = '../SFTP/'+info_servei['code']
    # Com que cada servei necessita uns paràmetres diferents els haurem d'executar per separat
    if(info_servei['id'] == 'APARCAR'):
        # El servei aparcar només necessita el ID del agent
        # Execucio del codi del servei
        output = subprocess.getoutput("sudo python3 "+codi_servei+" "+agentID)
        newJson = json.dumps(output)
        d = ast.literal_eval(output)

    if(info_servei['id'] == 'computar'):
        # El servei computar necessita extraparams (versio python, si utilitza PP o no, qui executa el servei)
        codi_a_computar = extraParams['nom_fitxer']
        versio = extraParams['version']
        pp = extraParams['paralel']
        quiExecuta = extraParams['quiExecuta']
        # Execucio del codi del servei
        d = subprocess.getoutput("sudo python3 "+codi_servei+" "+agentID+" "+codi_a_computar+" "+versio+" "+pp+" "+quiExecuta)
    return d
```

Figura 31: funció del mòdul RT per executar els serveis

7.3.6 API

Com en els anteriors casos, la API també correspon a una classe de Python amb els seus atributs i mètodes.

A l'inicialitzar la API li haurem de dir a quin objecte agent pertany, a quina adreça IP escoltarà i a quin port. Llavors, la classe API tindrà com a atributs (figura 32) els tres paràmetres mencionats i també totes i cadascuna de les bases de dades (col·leccions de una base de dades MongoDB) de la base de dades topològica i també de la base de dades del pàrquing.

```
@cherrypy.expose
class API:
    ########################## ATTR ######################################
    IP_DB = 'localhost'
    PORT_DB = 27017

    ########################## INIT API ##################################

    def __init__(self, agent, host=get_ip_address(), port=int(sys.argv[1])):
        self.agent = agent
        self.host = host
        self.port = port
        client = pymongo.MongoClient(self.IP_DB, self.PORT_DB)
```

Figura 32: atributs de la classe API

Cada mètode de la classe API serà una ruta a la qual podrem fer peticions HTTP. Per exemple, si el nom del mètode és function() i la IP i port de l'agent són 10.0.6.50 i 8000 respectivament, podrem fer peticions GET i POST a http://10.0.6.50:8000/function i la API ens contestarà segons el codi de la funció function(). La figura 33 mostra en termes de codi el que s'ha explicat en aquest paràgraf:

```
# Permet que li arribin peticions per aquesta funció
@cherrypy.expose
# Indica que li arriba un json com argument
@cherrypy.tools.json_in()
# Indica que retorna un json
@cherrypy.tools.json_out()
# Definició de la funció
def function(self):
    if cherrypy.request.method == "GET":
        # Aquí anirà el codi per a una petició GET
    if cherrypy.request.method == "POST":
        # Aquí anirà el codi per a una petició POST
```

Figura 33: exemple de funció de la API

7.3.7 Virtualització d'agents amb Docker

Per a la virtualització dels agents s'han utilitzat contenidors Dockers amb una imatge Ubuntu. A la imatge se li han instal·lat els paquets bàsics com SSH, Python i llibreries pertinents per poder córrer els agents.

Per tal de fer més senzilla la virtualització, s'ha fet un programa en Python 3 que arrenca n contenidors amb la imatge creada. Així, executant un sol programa Python ja en tenim prou per arrencar múltiples agents.

En el següent tros de codi mostra com s'arrenquen els contenidors. En Python podem utilitzar la llibreria "os" per a executar comandes GNU/Linux.

Per arrencar un contenidor, utilitzem la següent ordre (figura 34):

$ sudo docker run –dit –name <Nom contenidor> <ID de la imatge>

On el flag –dit serveix per a executar el contenidor en segon pla, <Nom contenidor> és el nom que li volem donar i <ID de la imatge> és l'identificador de la imatge a partir de la qual volem els contenidors (en aquest cas la Ubuntu mencionada anteriorment).

```
if __name__ == '__main__':
    for i in range(int(sys.argv[1])):
        os.system("docker run -dit --name My"+str(i)+" 9ea9d6091a89")
```

Figura 34: arrencar contenidors Docker des de Python emprant la llibreria os

Per a obtenir l'adreça IP del contenidor, emprem la comanda de la següent figura:

```
for i in range(int(sys.argv[1])):
    llista.append(os.popen("docker inspect --format '{{ .NetworkSettings.IPAddress }}' My"+str(i)).read())
```

Figura 35: obtenció de l'adreça IP d'un contenidor Docker des de Python

7.4 Implementació del panell d'administració

Per explicar la implementació de l'aplicació s'inclouran els troços de codi més rellevants i captures de pantalla del front-end per veure'n en resultat.

7.4.1 Plantilla general

Com moltes altres aplicacions web d'avui en dia, el front-end es basa en una plantilla comuna per a tots els mòduls. Tal i com s'ha explicat en l'anterior capítol, aquesta plantilla consta d'una barra lateral de navegació, que serà igual per a totes les vistes/mòduls de l'aplicació.

A més, diposa d'una capçalera (header) de color blanc amb un botó que conté el signe ☰, corresponent a l'icona de menú lateral. Aquest botó amaga la barra lateral si aquesta es troba desplegada o la mostra si es troba amagada.

Abaix de la capçalera blanca i al costat dret de la barra lateral és on situarem el contingut de cada mòdul.

Aquest és el resultat de la implementació de la plantilla (figura 36), feta a partir del disseny proposat a l'apartat 6.4.3 d'aquesta memòria.

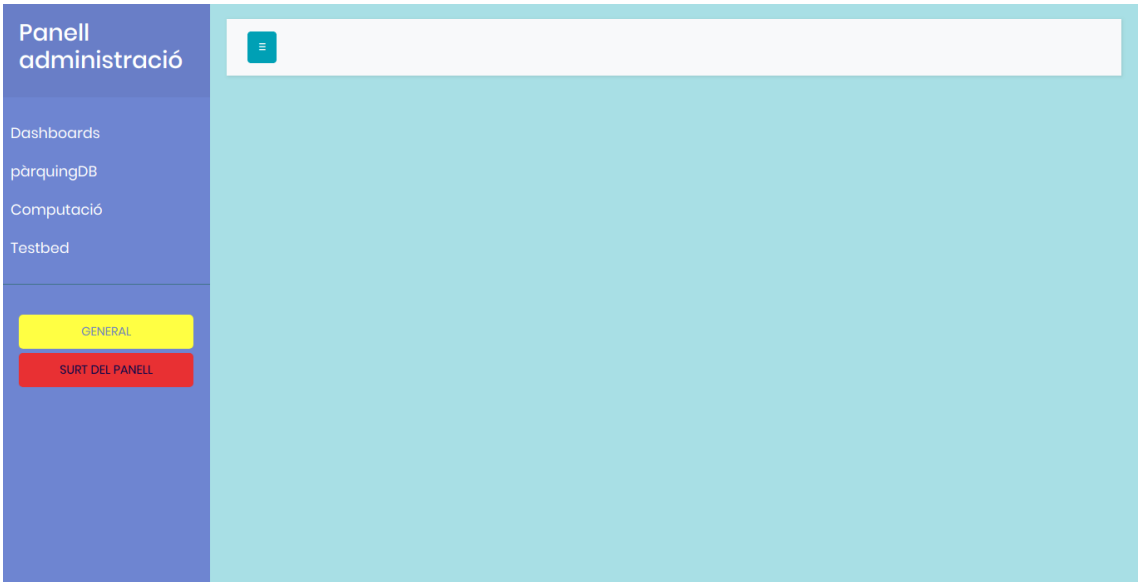


Figura 36: plantilla de l'aplicació panell d'administració

La següent imatge (figura 37) mostra el fragment de codi que correspon al contingut del mòdul.

HTML incorpora una eina per a dur a terme aquest tipus de plantilles.

Només escrivint {{{ body }}} , a cada vista de cada mòdul del front-end es mostrarà el contingut corresponent al codi definit al fitxer del mòdul en qüestió.

```
58
59          <!-- Page Content  -->
60          <div id="content">
61
62              {{{ body }}}
63
64          </div>
```

Figura 37: obtenció del cos (body) de cada mòdul en HTML

7.4.2 Mòdul d'inici

Consta de un formulari format per dos camps: nombre de plantes i nombre de places per planta. Aquests valors no poden ser qualsevol ja que sinó a l'hora de pintar l'esquema del pàrquing les línies no quadren i per tant l'esquema no és correcte. Per tal de que l'usuari entri un valor correcte, farem que aquests dos camps siguin cadascun una llista amb els possibles valors de cada camp. D'aquesta manera l'usuari només podrà entrar valors vàlids. La figura següent mostra el formulari en termes de codi HTML:

```
<label>Nombre de places per planta  <input list="browsers" id="input1"></label>

<datalist id="browsers">
  <option value="4">
  <option value="8">
  <option value="16">
  <option value="32">
  <option value="64">
  <option value="88">
  <option value="128">
  <option value="176">
  <option value="256">
  <option value="352">
  <option value="512">
</datalist>

<label>Nombre de plantes <input list="browsers2" id="input2"></label>

<datalist id="browsers2">
  <option value="1">
  <option value="2">
  <option value="3">
  <option value="4">
  <option value="5">
  <option value="6">
</datalist>
```

Figura 38: formulari d'entrada per a poguer seleccionar el nombre de places per planta i el nombre de plantes

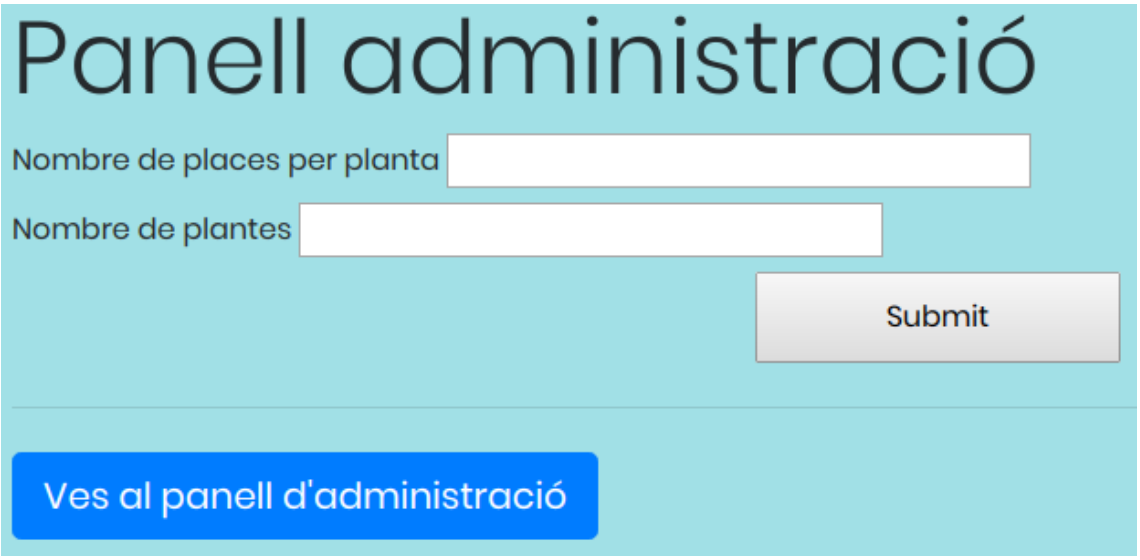


Figura 39: resultat gràfic del codi de la figura 40

Quan l'usuari entra els valors i fa clic al botó "Submit" se li mostra un avís de que el pàrquing s'ha configurat correctament amb els valors entrats. Aquest botó el que fa és una petició POST a la API del leader del pàrquing per tal de que actualitzi els valors a la base de dades del pàrquing.

Un cop configurat podrà anar al panell d'administració fent clic al botó blau, mostrat a la següent figura:

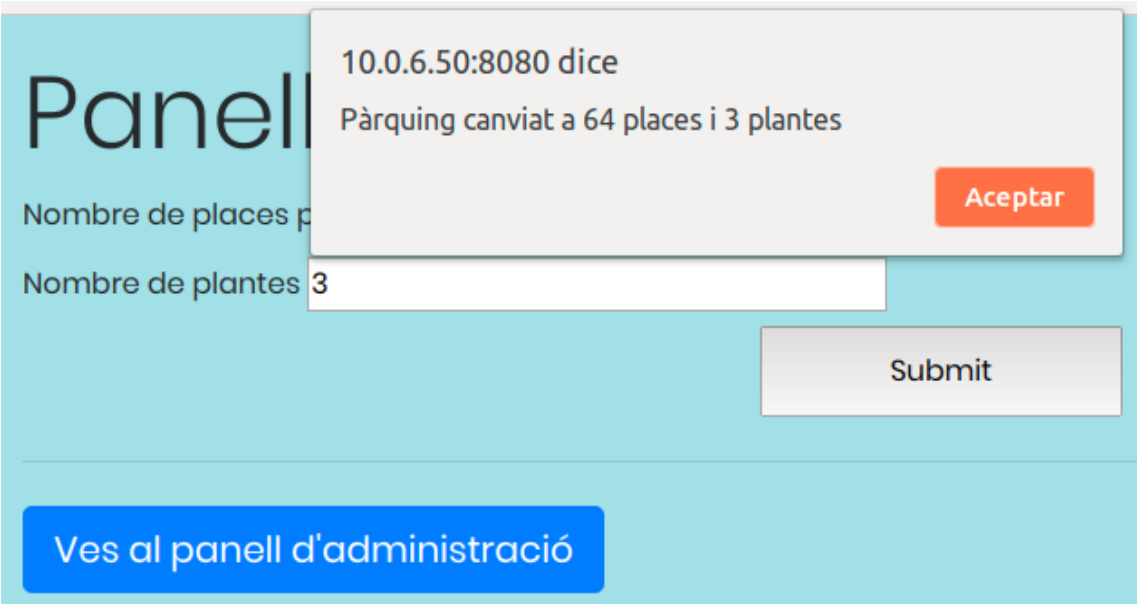


Figura 40: configuració del pàrquing feta correctament

7.4.3 Mòdul de computació

El mòdul de computació el podem dividir en dues grans parts: un formulari a omplir per l'usuari i una taula amb els fitxers i la seva informació.

```
<div id="form-wrapper">
  <form action="/upload" method="POST" enctype="multipart/form-data">
    <div class="form-group">
      Selecciona un fitxer Python per enviar-lo a computar:
      <input type="file" name="image">
    </div>

    <div class="form-group">
      <label for="inputVersio">Selecciona la versió de Python:</label>

      <input type="text" id="inputVersio" name="txtVersion" list="browsersPython">
      <datalist id="browsersPython">
        <option value="Python2">
        <option value="Python3">
      </datalist>
    </div>

    <label for="inputPP">Utilitza ParalelPython?</label>

    <input type="text" id="inputPP" name="txtPP" list="browsersPP">
    <datalist id="browsersPP">
      <option value="SÍ">
      <option value="NO">
    </datalist>
    <button id="btnsubmit" type="submit" class="btn btn-primary">Enviar</button>
  </form>
</div>
```

Figura 41: formulari a omplir per enviar un fitxer a computar

En la captura anterior (figura 41) es veu el codi HTML corresponent al formulari. Realment el que estem fent aquí és un POST a http://10.0.6.50:8080/upload amb la informació que l'usuari ha entrat.

Pel que fa a la taula, el codi és el següent:

```
<table align="center" id="MyTable" class="table">
  <thead>
    <tr>
      <th style="width: 2%">#</th>
      <th style="width: 5%">Nom del fitxer</th>
      <th style="width: 5%">Versió</th>
      <th style="width: 5%">Paral·lel</th>
      <th style="width: 5%">Estat</th>
      <th style="width: 5%">Data d'enviament</th>
      <th style="width: 5%">Data finalització</th>
      <th style="width: 5%">Clica sobre el nom per descarregar fitxer d'ouput</th>
      <th style="width: 2%">Esborra</th>
    </tr>
  </thead>

  <tbody>
   {{#each listfitxers}}
      <tr>
        <td>{{@index}}</td>
        <td>{{nom_fitxer}}</td>
        <td>{{version}}</td>
        <td>{{paralel}}</td>
        <td>{{estat}}</td>
        <td>{{data_entrada}}</td>
        <td>{{data_finalitzacio}}</td>
        <td style="cursor: pointer;">{{output}}</td>
        <td onclick="esborra_fitxer()" style="cursor: pointer; color: red; font-size: 30px;"><i class="fa fa-close"></i></td>
      </tr>
   {{/each}}
  </tbody>
</table>
</div>
```

Figura 42: codi HTML corresponent a la taula de fitxers computats o actualment en computació

Finalment el mòdul complet ha quedat de la següent manera:

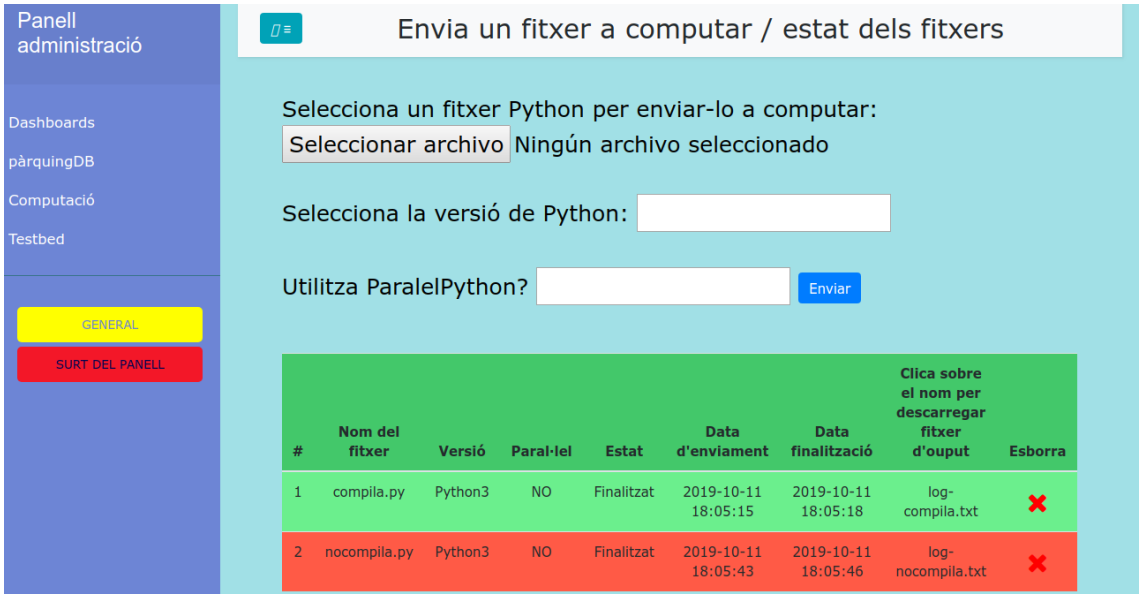


Figura 43: mòdul de computació

7.4.4 Mòdul llistat de vehicles

El mòdul serà accessible fent un GET des d'un navegador a la ruta http://10.0.6.50:8080/llista_vehicles , on 10.0.6.50 és l'adreça IP del front-end i escolta al port 8080.

Es composa principalment d'una taula HTML amb els camps de cada vehicle ja esmentats a l'anterior capítol. El codi de la taula en HTML és el següent:

```
<table id="MyTable" class="table">
  <thead>
    <tr>
      <th style="width: 2%">#</th>
      <th style="width: 5%">ID</th>
      <th style="width: 5%">Matrícula</th>
      <th style="width: 5%">Computa</th>
      <th style="width: 5%">Pagat</th>
      <th style="width: 5%">Plaça</th>
      <th style="width: 5%">IP agent</th>
      <th style="width: 5%">Recursos</th>
      <th style="width: 8%">Data entrada</th>
    </tr>
  </thead>

  <tbody>
   {{#each listvehicles}}
      <tr>
        <td>{{@index}}</td>
        <td>{{_id}}</td>
        <td>{{matricula}}</td>
        <td>{{computa}}</td>
        <td>{{pagat}}</td>
        <td>{{plasa}}</td>
        <td>{{agentIP}}</td>
        <td>{{resources}}</td>
        <td>{{data_entrada}}</td>
      </tr>
   {{/each}}
  </tbody>
</table>
```

Figura 44: codi HTML de la taula que conté la informació de tots els vehicles estacionats al pàrquing

La següent captura de pantalla (figura 45) mostra com queda la taula visualment:

Figura 45: resultat del codi de la figura 44

Per a que l'administrador pugui localitzar un vehicle ràpidament, disposa d'un filtre situat a la part superior dreta de la pantalla (figura 46), on pot escriure part d'una matrícula o una matrícula sencera i li apareixeran a la taula només els vehicles que coincideixin amb el text entrat.

Figura 46: resultat de l'aplicació d'un filtratge per matrícula

7.4.5 Mòdul vista general

El mòdul serà accessible fent un GET des d'un navegador a la ruta http://10.0.6.50:8080/admin , on 10.0.6.50 és l'adreça IP del front-end i escolta al port 8080.

El dibuix del pàrquing s'ha fet amb l'element canvas que incorpora el llenguatge HTML versió 5.

En aquest cas li he donat unes dimensions de 1000 píxels d'amplada i 500 píxels d'alçada (figura 47).

```
<!-- Canvas -->
<div id=wrapper>
  <canvas id="canvas1" width=1000 height=500 style="border:4px solid black;"></canvas>
</div>
```

Figura 47: declaració d'un rectangle de 1000x500 píxels amb l'element canvas

A continuació es mostra una captura de pantalla (figura 48) d'aquest mòdul, en les seves condicions inicials (sense cap filtre aplicat):

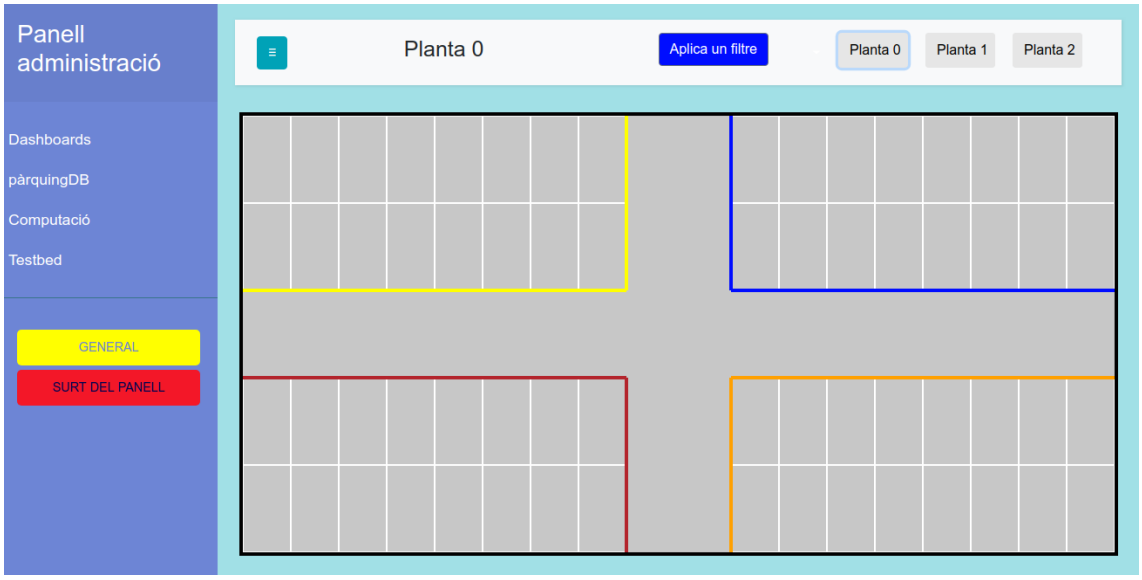


Figura 48: visualització gràfica inicial del pàrquing

Es va decidir fer cada planta del pàrquing rectangular per temes de simplicitat i ajustament a la pantalla, ja que quasi totes són rectangulars o semblants. A més, s'han buscat les dimensions per a que no s'hagi de fer *scroll* (baixar la pàgina) per a veure la totalitat del dibuix.

Així, sigui quina sigui la mida de cada planta, podem dividir el rectangle en 4 zones ben definides, segons la posició on es troben:

Zona North West (NW) → zona superior esquerra del rectangle, marcada en color groc.

Zona North East (NE) → zona superior dreta del rectangle, marcada en color blau.

Zona South West (SW) → zona inferior esquerra del rectangle, marcada en color vermell.

Zona South East (SE) → zona inferior dreta del triangle, marcada en color taronja.

Tal i com es va definir a l'anterior capítol, quan l'usuari fa clic sobre una plaça del pàrquing al dibuix, se li mostra un quadre de text amb la informació més rellevant del vehicle/agent.

Podem detectar quan l'usuari fa clic a sobre del canvas afegint-li un event anomenat "mousedown" al canvas a l'hora de dibuixar-lo, tal i com mosta la següent figura:

```
function draw1(){
  c = document.getElementById("canvas1");
  var width = c.width;
  var height = c.height;
  c.addEventListener('mousedown', onDown1, false);
  ctx = c.getContext("2d");
```

Figura 49: funció javascript per detectar quan l'usuari fa clic sobre el gràfic del pàrquing

Així, cada vegada que l'usuari cliqui amb el ratolí sobre la superfície del canvas, s'executarà la funció onDown1. Dins d'aquesta funció anirà el codi que s'encarregarà de fer les consultes pertinents per obtenir la informació del vehicle que està aparcat en la plaça clicada.

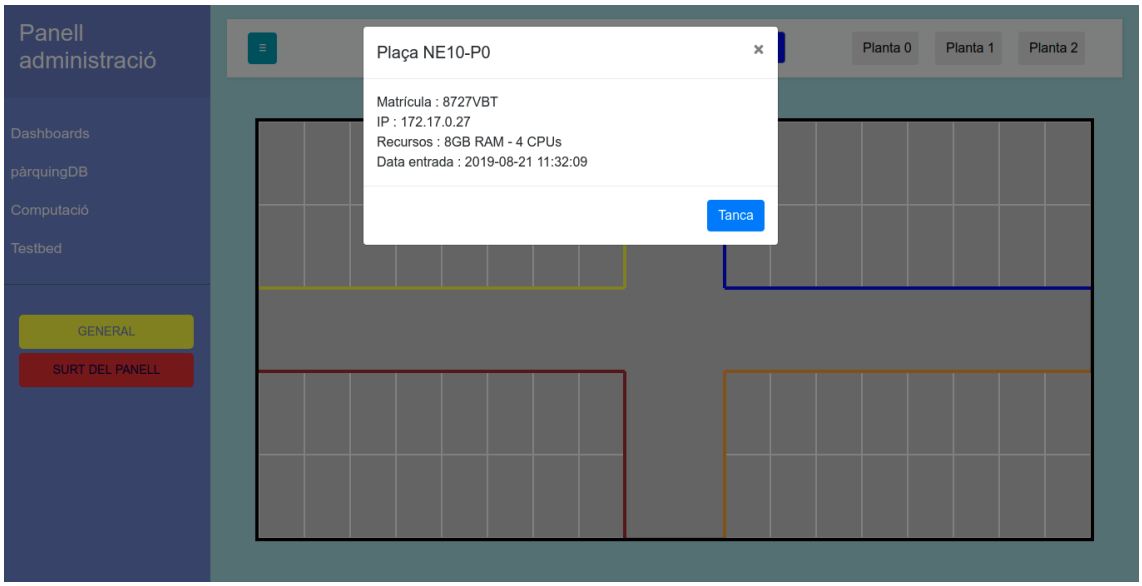


Figura 50: resultat de fer clic a la plaça NE10 de la planta 0

Filtre d'ocupació

Si fem clic al botó blau "Aplica un filtre", se'ns desplega un llistat amb els dos tipus de filtre que podem seleccionar. El primer d'ells es tracta del filtre "Ocupació", que ens pinta de color vermell les places que estan ocupades i de color verd les que estan lliures (figura 51).

Figura 51: resultat d'aplicar el filtratge d'ocupació

Filtre de computació

L'altre dels filtres disponibles és el de computació. Semblant al cas anterior, però ara se'ns pintaran de color verd fort les places que el vehicle formi part del clúster de computació (figura 52).

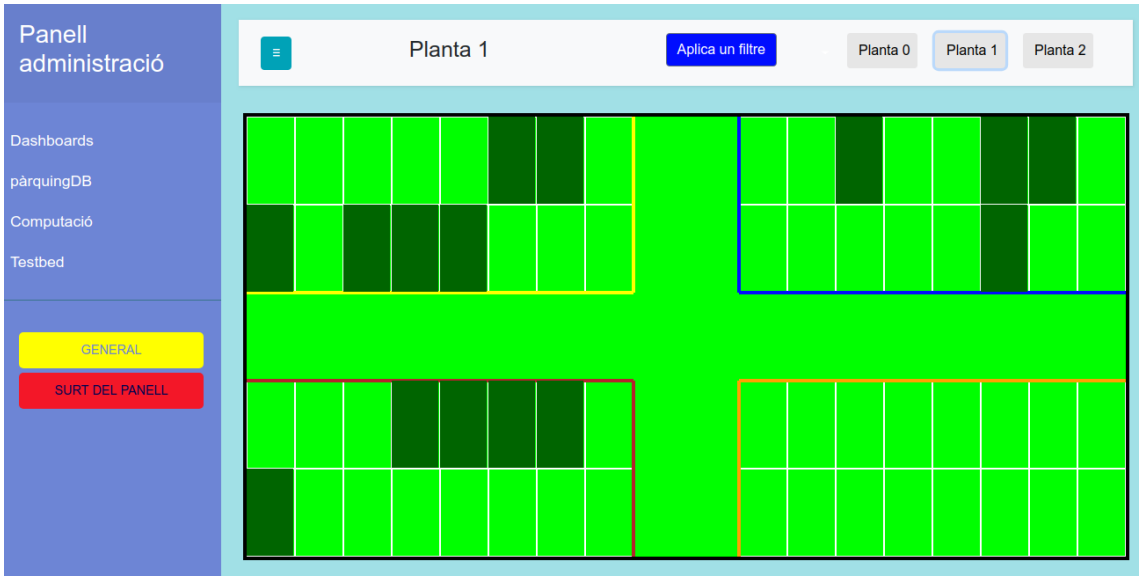


Figura 52: resultat d'aplicar el filtratge de computació

7.4.6 Mòdul dashboards

Per a la implementació de cada gràfic del mòdul s'ha utilitzat la llibreria Chart.js.

Cada gràfic correspon a una funció Javascript. En cada funció es fa una petició GET emprant el client HTTP per a javascript anomenat axios. Aquestes peticions es fan a la API de l'agent leader del pàrquing. Llavors, utilitzem les dades/informació de la resposta de la petició per a fer els gràfics tal i com mostra la figura 53 (variable humy en el codi).

```
function display_reserves() {
  axios.get('http://'+ipLeader+':8000/update_reserves').then(response => {
    let humy = response.data
    let myChart = document.getElementById('myChart6').getContext('2d');
    let reservesChart = new Chart(myChart, {
      scaleOverride : true,
      type:'bar',
      data:{
        labels:['Reserves'],
        datasets:[{
          label:'Nº reserves per avui',
          data:[
            humy
          ],
          backgroundColor:[
            'purple',
          ],

        }]
      },
      options:{
        scales: {
            yAxes: [{
                ticks: {
                    precision : 0,
                    beginAtZero: true,
                    max : 4,
                    stepSize: 32
                }
            }]
        },
```

Figura 53: codi javascript corresponent al gràfic de reserves

En aquest cas la variable humy conté el nombre de reserves que hi han actualment. Llavors, creem el gràfic, li indiquem que és del tipus 'bar' (gràfic de barres), tindrà una etiqueta 'Reserves' i de color lila.

Aquesta llibreria ens permet ajustar els paràmetres del gràfic tals com els valors dels eixos X i Y.

A continuació podem veure una imatge de com ha quedat el mòdul finalment:

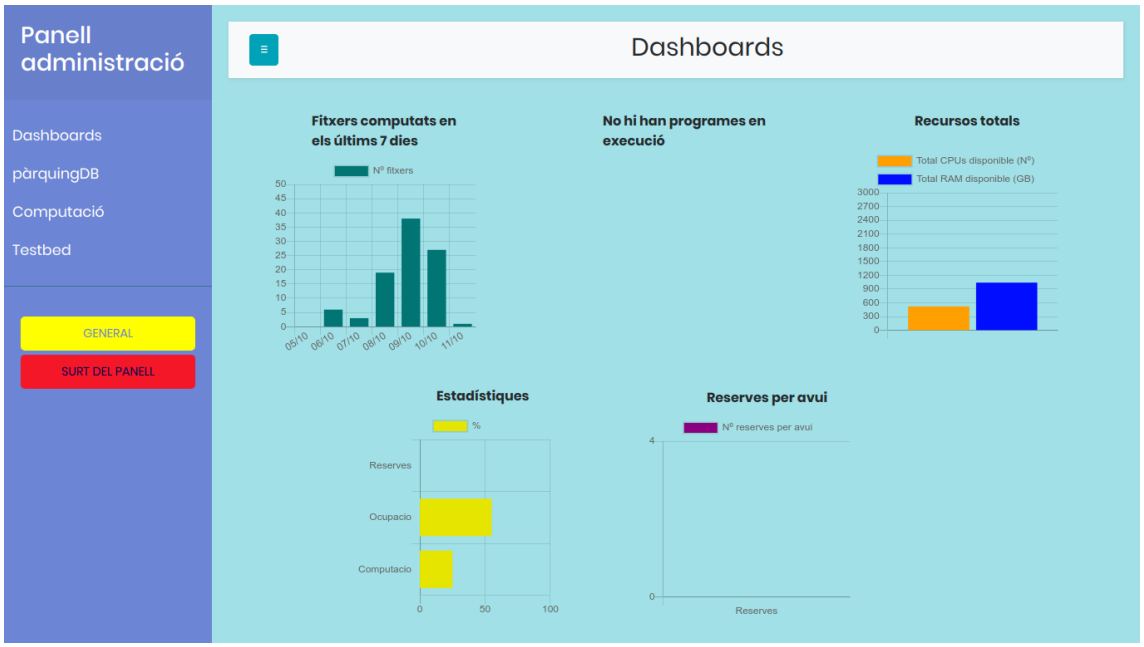


Figura 54: mòdul dashboards del panell d'administració

7.4.7 Mòdul Testbed

El mòdul Testbed consta d'un canvas semblant al que hem vist a la vista general, però aquest segueix l'esquema del pàrquing del testbed. La mida del canvas és de 800x600.

Al fer clic sobre cada plaça, si està ocupada, mostra la informació del vehicle. Com es pot apreciar a la següent captura de pantalla (figura 55), les places ocupades es mostren de color vermell.

La implementació és molt semblant en quant a codi al mòdul vista general, però ara tenim un nombre de places definit i per tant és més fàcil tenir controlades les places.

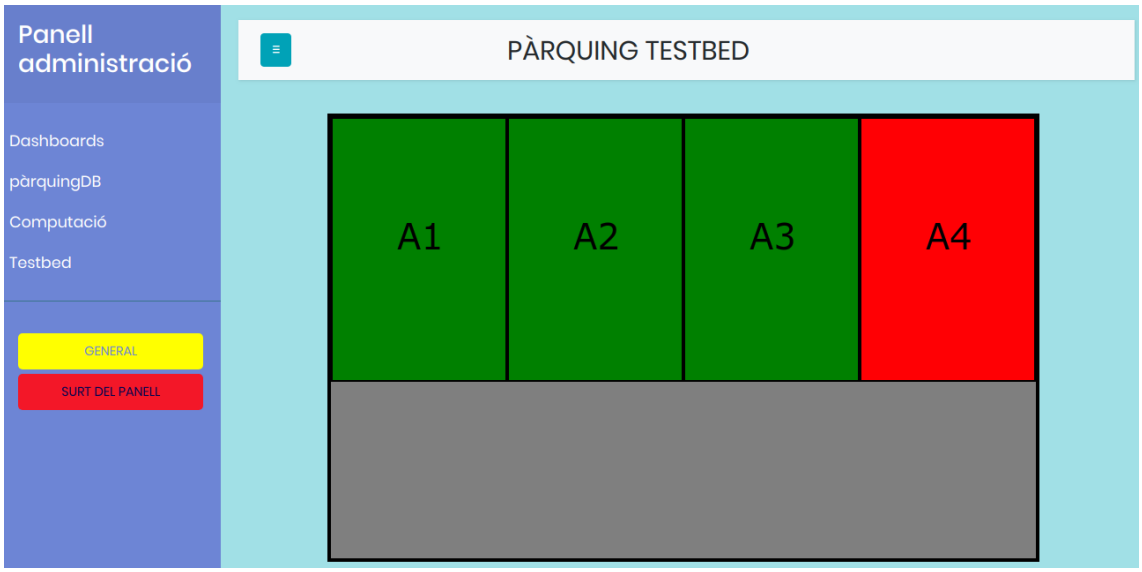


Figura 55: mòdul testbed del panell d'administració

7.5 Aplicació client

S'han utilitzat dos botons del framework web Bootstrap per a demanar els serveis. Aquests dos botons corresponen als que podem observar a la figura 56.

En el cas del servei d'aparcament, al fer clic al botó s'envia una petició POST a l'agent del cotxe (el servei en detall s'explica a l'apartat 9.2).

Pel que fa al servei de computació i com s'ha comentat anteriorment al capítol de disseny, al fer clic sobre botó ens portarà a una altra pantalla (figura 57) on podrem sol·licitar el servei (ja que l'usuari haurà d'omplir un formulari).

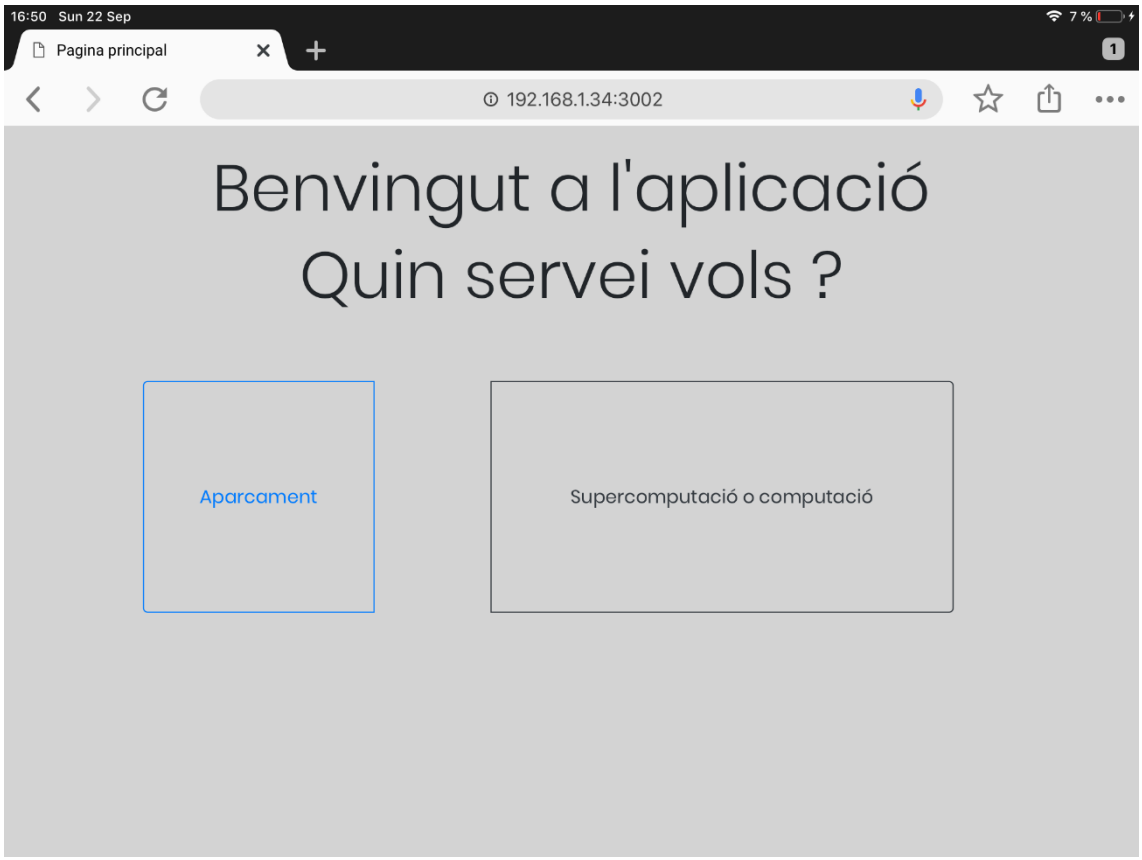


Figura 56: pantalla inicial de l'aplicació client

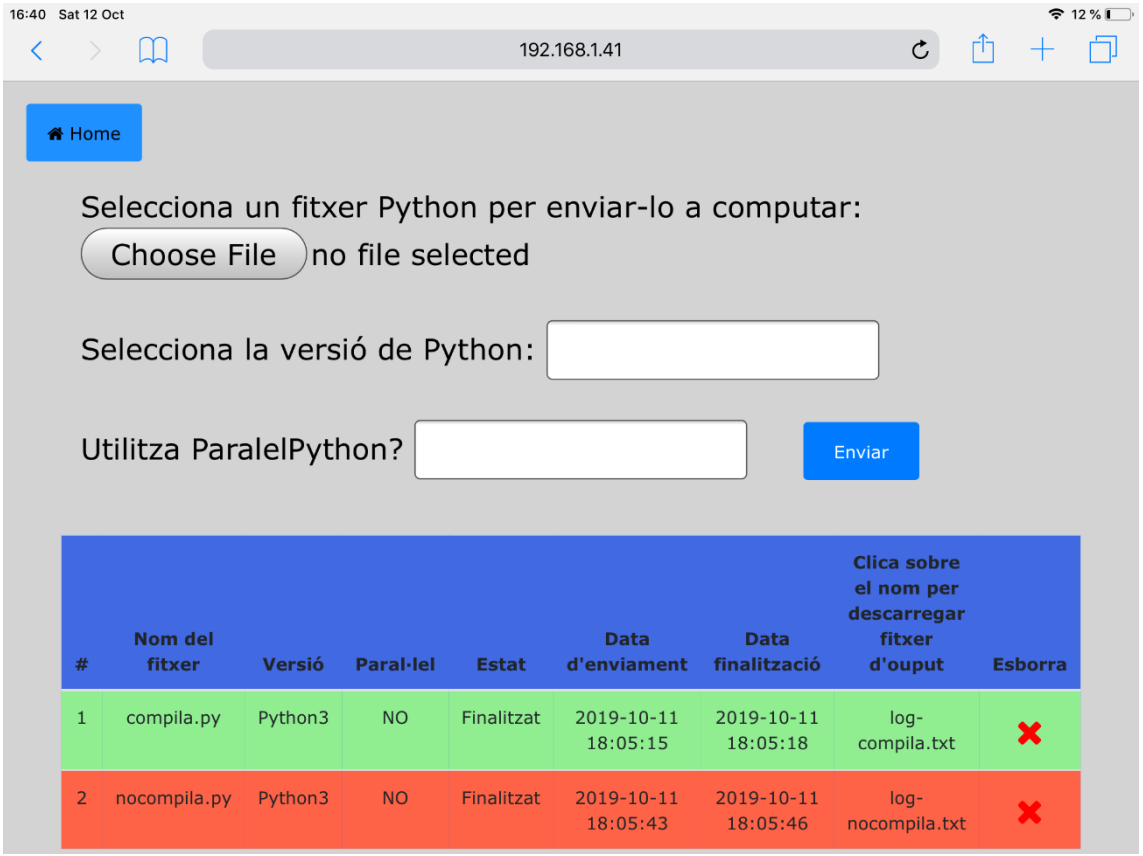


Figura 57: pantalla per a demanar el servei de supercomputació o computació de l'aplicació del client

Si l'usuari fa clic sobre el botó blau "Home" de l'anterior figura serà retornat a la pantalla inicial.

CAPÍTOL 8 – SERVEIS

Arribats a aquest punt ja s'ha explicat tota la arquitectura que utilitzarem i els elements que la composen.

És evident que tot això ho estem fent per aportar utilitats i satisfer als nostres clients i usuaris.

Els apartats que segueixen són imprescindibles i són la clau per entendre aquest projecte i la seva finalitat, més enllà dels agents i tota la infraestructura que s'ha explicat fins ara.

8.1 Model de negoci

Podem dir que tenim dos tipus ben diferents de **clients**:

1. El que vol executar un programa o servei o aplicació al nostre clúster de màquines (pàrquing) en paral·lel o no. Aquest client pot ser un node de la ciutat intel·ligent, per exemple un hospital que necessita córrer un servei o bé una entitat externa com per exemple una empresa.
2. La persona que vol aparcar el seu vehicle al nostre pàrquing.

Client tipus 1

Podem dividir aquest tipus de client en dos subtipus:

- El primer és una entitat que vol executar un programa aprofitant la nostra infraestructura de computació formada per els nodes del pàrquing.
- El segon és un element/entitat que pertany a la ciutat intel·ligent i que necessita córrer serveis que requereixin una baixa latència. Un exemple d'aquest tipus podria ser un hospital que necessita calcular una ruta per a una ambulància.

En els dos subtipus es suposa que el client té coneixements informàtics, ja que haurà de saber com enviar el fitxer i probablement l'haurà escrit ell (una persona que no té coneixements de Python rarament voldrà enviar un fitxer a computar).

Client tipus 2

Aquest client és el que vol estacionar el seu vehicle al nostre pàrquing i cedir-nos els recursos computacionals del seu agent per a que el seu vehicle formi part del clúster de computació on s'executaran els serveis.

També existeix la possibilitat de que, per algun motiu, el client no vulgui cedir-nos els seus recursos, és a dir, que només vulgui estacionar.

En el cas de sí que ho faci, li oferirem algun tipus de bonificació (per exemple, un % del temps que ha estat estacionat el seu vehicle serà gratuït).

Notem que aquest client no té coneixements informàtics (en principi) com en el cas anterior, i per tant haurem de fer el nostre model d'aparcament el més sencill i fàcil possible d'utilitzar. Aquest requisit el compleix la nostra aplicació per als clients, doncs **només haurà de fer clic a un botó per a demanar-lo**.

8.2 Serveis oferts

En aquest projecte s'han dissenyat i implementat dos serveis:

- Servei de supercomputació o computació (depenent si el programa a computar utilitza Parallel Python o no).
- Servei d'aparcament intel·ligent.

8.2.1 Servei de computació

Aquest servei permet enviar un programa (un fitxer Python) a computar al conjunt de màquines (clúster) que tenim en un pàrquing.

La petició del servei es fa des l'aplicació front-end destinada al client que permetrà pujar un fitxer que el client tingui en la seva màquina local a computar-se, i un cop finalitzada l'execució, obtenir-ne el resultat.

En el cas de que fos un element de la ciutat intel·ligent que necessiti enviar un fitxer a computar, ho farà fent una petició POST a la API de l'agent leader del pàrquing. En aquesta petició haurà d'incloure el fitxer a computar, si utilitza Parallel Python o no, i quina versió de Python és. El resultat s'obtindrà de la mateixa manera que abans, amb l'aplicació front-end.

La figura 58 mostra el diagrama del servei pas a pas:

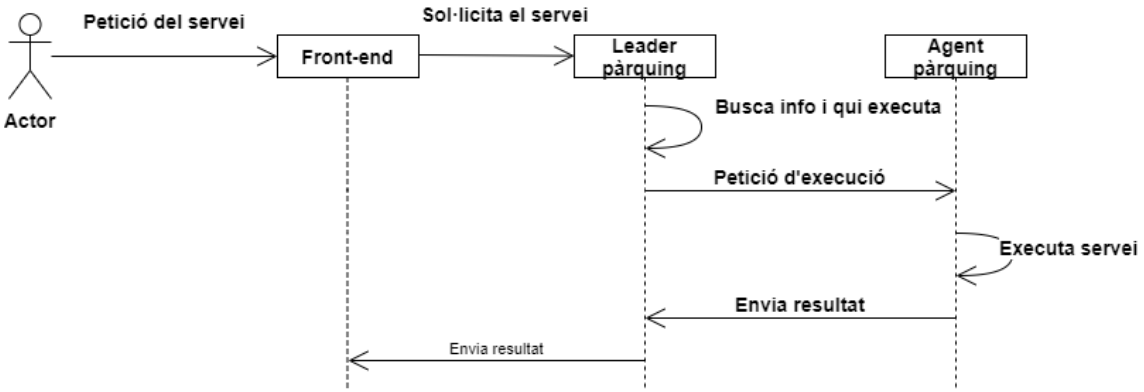


Figura 58: diagrama de comunicacions del servei de supercomputació o computació

Hi ha un escenari que hem de contemplar (i més en el cas d'un pàrquing on tenim vehicles que entren i surten a cada moment): què passa si l'agent encarregar d'executar el servei cau o simplement marxa del pàrquing?

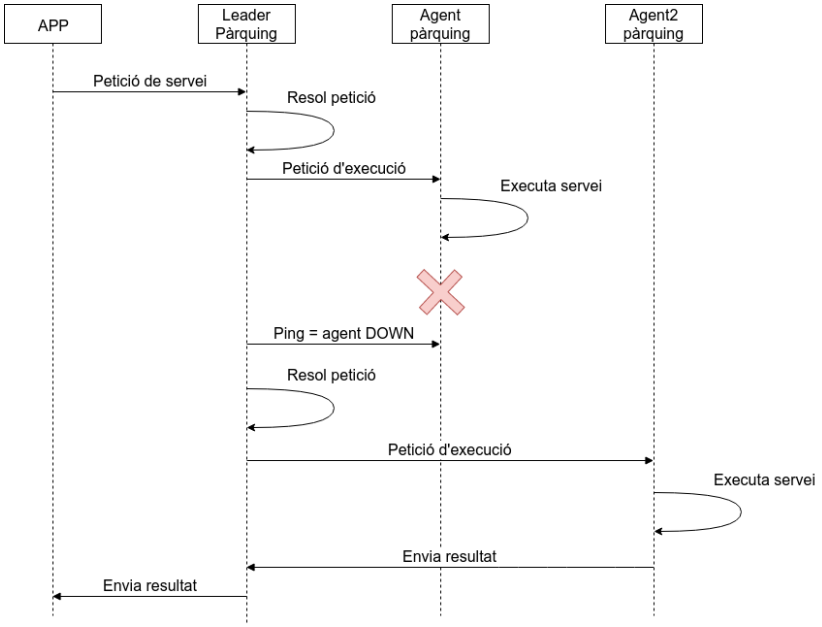


Figura 59: diagrama de comunicacions en cas de que l'agent encarregat d'executar el servei falli

En el cas de que l'agent encarregat caigui o per algun motiu no completi l'execució del servei, l'agent leader se'n adonarà ja que l'agent deixarà d'executar el fitxer i li enviara un resultat buit o directament no li enviarà (si marxa del pàrquing). Si això passa, l'agent leader enviarà un ping a l'agent encarregat per comprovar si està aixecat o no (figura 59). Si no ho està, el que

farà serà assignar-li la petició d'execució a un altre agent del pàrquing que es trobi actiu. Aquest procés és recursiu, és a dir, si el nou encarregat també cau l'agent leader li assignarà a un altre, i així successivament fins arribar a l'execució exitosa del servei. Si l'agent encarregat esta aixecat, vol dir que l'error és propi del codi (el client ha enviat un codi que no compila, per exemple) i per tant no hi podem res més que notificar-li al client a la seva aplicació.

8.2.2 Servei d'aparcament intel·ligent

El servei d'aparcament intel·ligent implementat permet al client estacionar el seu vehicle d'una manera eficaç, eficient i més ràpida ja que no haurà de perdre temps circulant per la ciutat i buscant una plaça per aparcar. Llavors, partirem d'un vehicle situat en algun punt de la ciutat i que el seu conductor (el nostre client) vol aparcar i només haurà d'obrir l'aplicació i sol·licitar el servei.

Aquest servei consta de 4 subserveis:

1. Localitzar el pàrquing més proper a la posició del vehicle que demana el servei. En el cas de que el pàrquing més proper estigui complet (no hi hagi plaça lliure) es buscarà el segon més proper, i així recursivament fins a trobar-ne un.
2. Reservar una plaça **qualsevol** del pàrquing localitzat per al agent que sol·licita el servei.
3. Càlcul de la ruta més curta al pàrquing, a partir de la posició del agent que demana el servei i la posició del pàrquing escollit. Un cop calculada la ruta se li envia al agent, i aquest la segueix fins arribar al pàrquing.
4. Quan el vehicle arriba al pàrquing, l'agent del vehicle envia un avís a l'agent leader del pàrquing i aquest li indica a quina plaça ha d'aparcar. Quan s'envia aquest avis, també s'envia la informació de l'agent, per tal de poguer tenir-lo controlat en tot moment (matrícula, recursos, si computa o no, etc…).

El servei d'aparcament es demana des de l'aplicació front-end per al client i s'ha implementat de forma conjunta amb un altre equip.

L'objectiu d'aquesta implementació conjunta és potenciar el servei fent que el cotxe del client que sol·licita el servei sigui autònom, és a dir, que no el condueixi ningú. Aquesta integració i els diagrames de comunicació del servei s'expliquen al següent capítol.

CAPÍTOL 9 – INTEGRACIÓ AMB ALTRES PROJECTES

9.1 Explicació dels altres projectes

Primerament es farà una breu explicació dels projectes externs i els aspectes que s'han integrat.

Notem que tots els projectes es basen en la mateixa arquitectura F2C amb l'element agent com a peça clau. Per tant, a partir d'aquesta arquitectura cada equip ha implementat els seus propis agents i els seus propis serveis.

9.1.1 Conducció autònoma i calibració

Aquest equip està treballant serveis que requereixen una conducció autònoma per part dels vehicles que participen en els serveis. També han implementat serveis de calibració d'una smart city, en tots dos casos al testbed del CRAAX.

De cara a la integració, però, no tindrem en compte la part de la calibració ja que no ens aporta res de cara a l'aparcament intel·ligent. Ens centrarem doncs en la part de conducció autònoma.

Per a dur a terme el seu projecte, han desenvolupat diversos serveis utilitzant el testbed i els seus vehicles. Un dels serveis que han implementat és la conducció autònoma d'un vehicle des d'un punt del testbed fins a un altre. Aquest és el servei que necessitem per a completar el servei d'aparcament intel·ligent, ja que en el nostre projecte no s'ha treballat amb els vehicles del testbed ni en càlculs de rutes mínimes.

9.2 Servei d'aparcament intel·ligent (conjuntament amb l'equip de conducció autònoma)

Estructura i agents involucrats (figura 60):

- Agent A
- Leader A
- Leader B
- Agent cloud

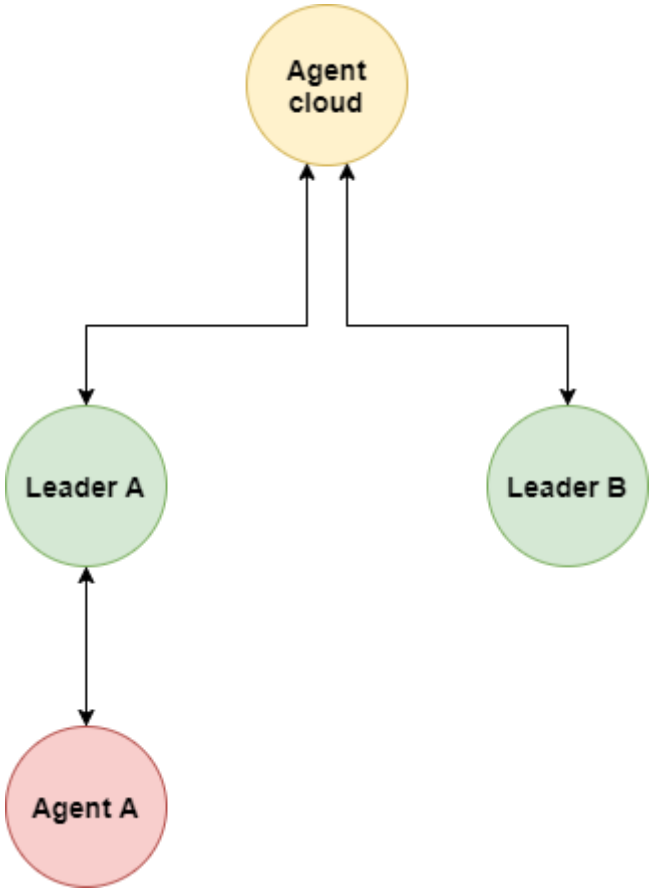


Figura 60: agents involucrats en el servei d'aparcament intel·ligent conjunt

Partim d'un vehicle situat en qualsevol punt de la ciutat (testbed). En un moment donat, el conductor d'aquest vehicle fa una petició del servei d'aparcament, mitjançant l'aplicació client.

Les accions que es segueixen per dur a terme el servei són:

Part 1 (figura 60):

1. El client fa la petició del servei d'aparcament des de l'aplicació. Aquesta petició es fa contra l'agent del seu vehicle, en aquest cas l'Agent A.
2. L'agent A delega la petició al seu leader, el leader A. Aquest comprova les dependències del servei. La primer dependència correspon a trobar el pàrquing més proper disponible i reservar-hi una plaça. Com que no pot (no té la capacitat de fer-ho) resoldre'l, delega la petició a l'Agent cloud.
3. El leader A executa serveis de gestió de semàfors, tràfic i fanals per tal de sincronitzar i manejar aquests elements a la ciutat.
4. L'Agent cloud busca qui pot resoldre aquest servei (en aquest cas el Leader B o leader del pàrquing) i li delega la petició d'execució.
5. El Leader B executa el servei: troba pàrquing més proper i reserva una plaça. Com a resultat d'aquest servei li envia a l'Agent cloud la posició del pàrquing on s'ha fet la reserva.
6. L'Agent cloud envia la posició al Leader A. Aquest busca la següent dependència del servei. Correspon al càlcul de ruta més curta des de la posició del vehicle sol·licitant al pàrquing. No ho pot fer ja que no té la posició on es troba el vehicle. Per tant li demana la posició a l'agent A.
7. L'agent A troba la seva posició dins la ciutat i li envia al leader A.
8. Quan el leader A té la posició ja pot fer el càlcul de la ruta mínima. Un cop calculada sol·licita a l'agent A que la segueixi per tal d'arribar al pàrquing.
9. El vehicle segueix la ruta fins al pàrquing.

Part 2 (figura 61):

1. Un cop el vehicle arriba a la posició final (posició del pàrquing) envia al Leader B la seva informació (matrícula, si computa o no).
2. El Leader B comprova si té reserva.
3. Com que té reserva, aquest li diu a quina plaça ha d'aparcar.

4. El vehicle va a la plaça assignada i quan hi arriba li envia una confirmació al leader conforme ja ha estacionat.
5. L'agent leader s'encarrega d'actualitzar el panell d'administració, en aquest cas el mòdul Testbed.

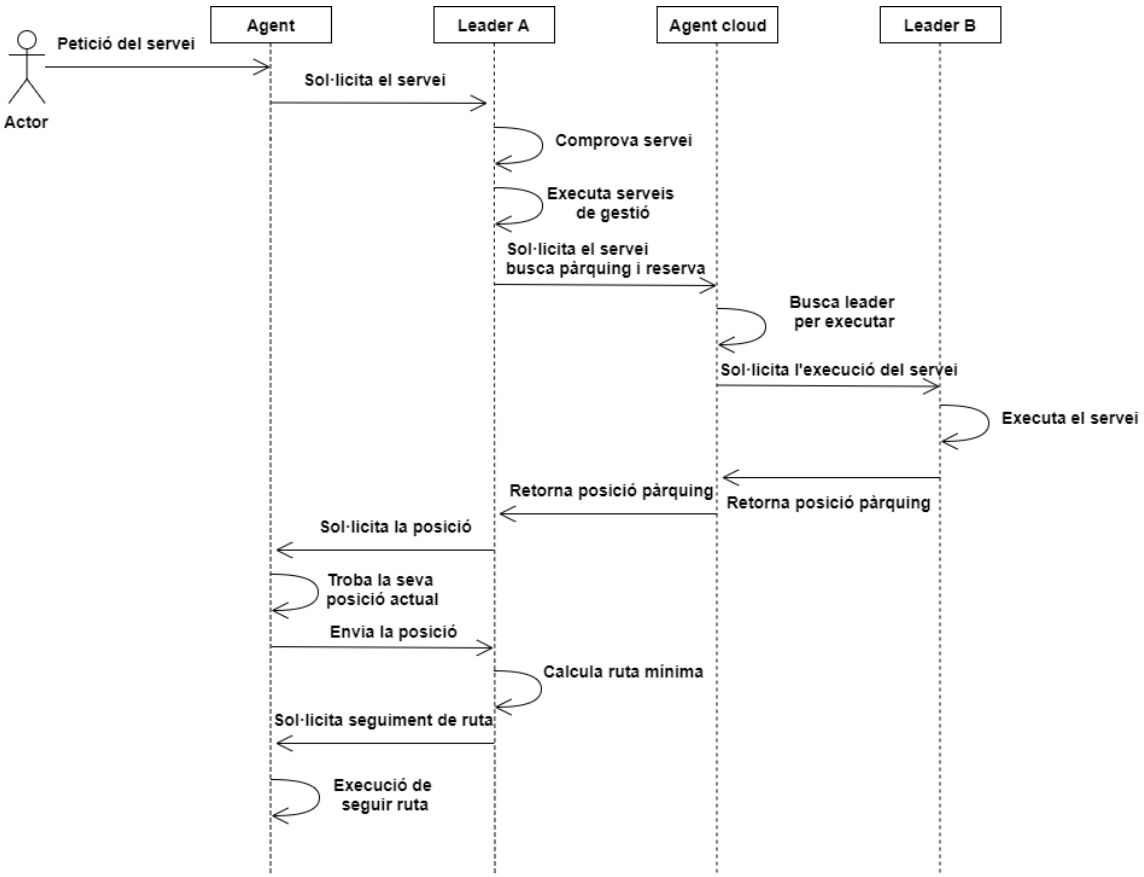


Figura 61: diagrama de comunicacions per al servei d'aparcament intel·ligent conjunt (part 1)

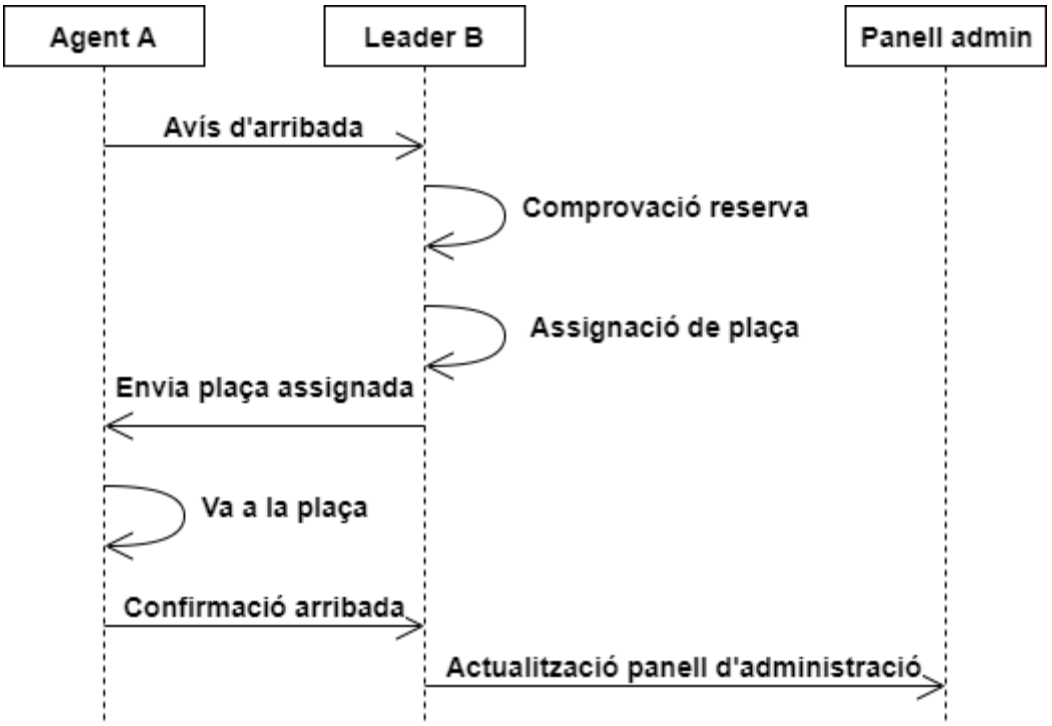


Figura 62: diagrama de comunicacions per al servei d'aparcament intel·ligent conjunt (part 2)

CAPÍTOL 10 – PROVES DE FUNCIONAMENT

Com en tot projecte, necessitem verificar el correcte funcionament de cadascun dels elements implementats per tal de veure si es comporten tal i com s'espera.

A continuació es presenten algunes de les proves fetes.

10.1 Test comunicació entre agents

Recordem que la comunicació entre els agents es fa a través de la seva API. En aquesta primera prova es fa una petició (GET) d'un agent a un altre, per veure que la petició és enviada i rebuda correctament.

10.1.1 Petició GET

Partint d'un agent ja arrancat i la seva API escoltant peticions al port 8000, fem una petició GET a un agent:

Petició: GET http://10.0.6.50:8000/get_leader_agents/192.168.1.38

Amb aquesta petició estem obtenint tots els agents que l'agent leader amb IP 192.168.1.38.

```
aerie@aerie-X550LD:~/Documentos/DEMO10/shar$ sudo python3 main_newagent.py 8000
[sudo] contraseña para aerie:
[02/Sep/2019:13:06:07] ENGINE Listening for SIGTERM.
[02/Sep/2019:13:06:07] ENGINE Listening for SIGHUP.
[02/Sep/2019:13:06:07] ENGINE Listening for SIGUSR1.
[02/Sep/2019:13:06:07] ENGINE Bus STARTING
CherryPy Checker:
The config entry 'tools.response.headers.headers' may be invalid, because the 'r
esponse' tool was not found.
section: [/]

[02/Sep/2019:13:06:07] ENGINE Started monitor thread 'Autoreloader'.
[02/Sep/2019:13:06:07] ENGINE Serving on http://10.0.6.50:8000
[02/Sep/2019:13:06:07] ENGINE Bus STARTED
10.0.6.50 - - [02/Sep/2019:13:06:11] "GET /get_leader_agents/192.168.1.38 HTTP/1
.1" 200 195 "" "PostmanRuntime/7.6.0"
```

Figura 63: petició GET a la API d'un agent

Resposta: [{"_id": "473", "host": "10.0.6.50", "port": 8000, "device": "AgentNil", "role": "agent", "IOT": "IOT", "leaderIP": "192.168.1.38", "status": 1, "nodeID": "b8eaf9ca-62ec-4ef6-ae5c-3004d138f48f"}]

Com veiem, ens retorna un array de diccionaris, on cada diccionari correspon a la informació d'un agent. En aquest cas el leader només té 1 agent connectat.

10.2 Test afegir agent a la topoDB

Amb un leader agent o l'agent cloud en marxa (recordem que només els agents amb alguna d'aquestes dues funcions pot accedir a la DB topològica), li fem una petició POST amb els camps necessaris. Ho podem fer des de Python amb la llibreria requests, tal i com es mostra al fragment de codi de la figura 64:

```
if __name__ == '__main__':
    leaderIP = '10.0.6.50'
    data = {
        'myIP' : '10.2.16.32',
        'port' : 8000,
        'device' : 'Agent prova',
        'role' : 'agent',
        'leaderIP' : leaderIP,
        'IoT' : ['IoT 1', 'IoT 2']
    }
    req  = requests.post('http://10.0.6.50:8000/register_agent', json=data)
```

Figura 64: petició POST per a enregistrar un agent a la base de dades topològica

No cal enviar-li el camp nodeID ja que l'agent leader ja li genera un al rebre la petició d'enregistrament. Haurem de veure (figura 65) com arriba la petició POST /register_agent al leader.

```
aerie@aerie-X550LD:~/Documentos/DEMO10/shar$ sudo python3 main_newagent.py 8000
[16/Sep/2019:19:13:07] ENGINE Listening for SIGTERM.
[16/Sep/2019:19:13:07] ENGINE Listening for SIGHUP.
[16/Sep/2019:19:13:07] ENGINE Listening for SIGUSR1.
[16/Sep/2019:19:13:07] ENGINE Bus STARTING
CherryPy Checker:
The config entry 'tools.response.headers.headers' may be invalid, because the 'r
esponse' tool was not found.
section: [/]

[16/Sep/2019:19:13:07] ENGINE Started monitor thread 'Autoreloader'.
[16/Sep/2019:19:13:07] ENGINE Serving on http://10.0.6.50:8000
[16/Sep/2019:19:13:07] ENGINE Bus STARTED
10.0.6.50 - - [16/Sep/2019:19:13:10] "POST /register_agent HTTP/1.1" 200 5 "" "p
ython-requests/2.18.4"
```

Figura 65: comprovació de que la petició POST arriba a l'agent leader

Finalment comprovem que s'hagi afegit correctament amb l'eina MongoDB Compass (figura 66):

Figura 66: comprovació de que l'agent s'ha enregistrat correctament a la base de dades topològica

10.3 Test del servidor i client web

Per verificar que el servidors de les aplicacions panell d'administració i aplicació del client estan en marxa i funcionant, ens fixem en els missatges que apareixen un cop executem l'ordre per arrancar els contenidors:

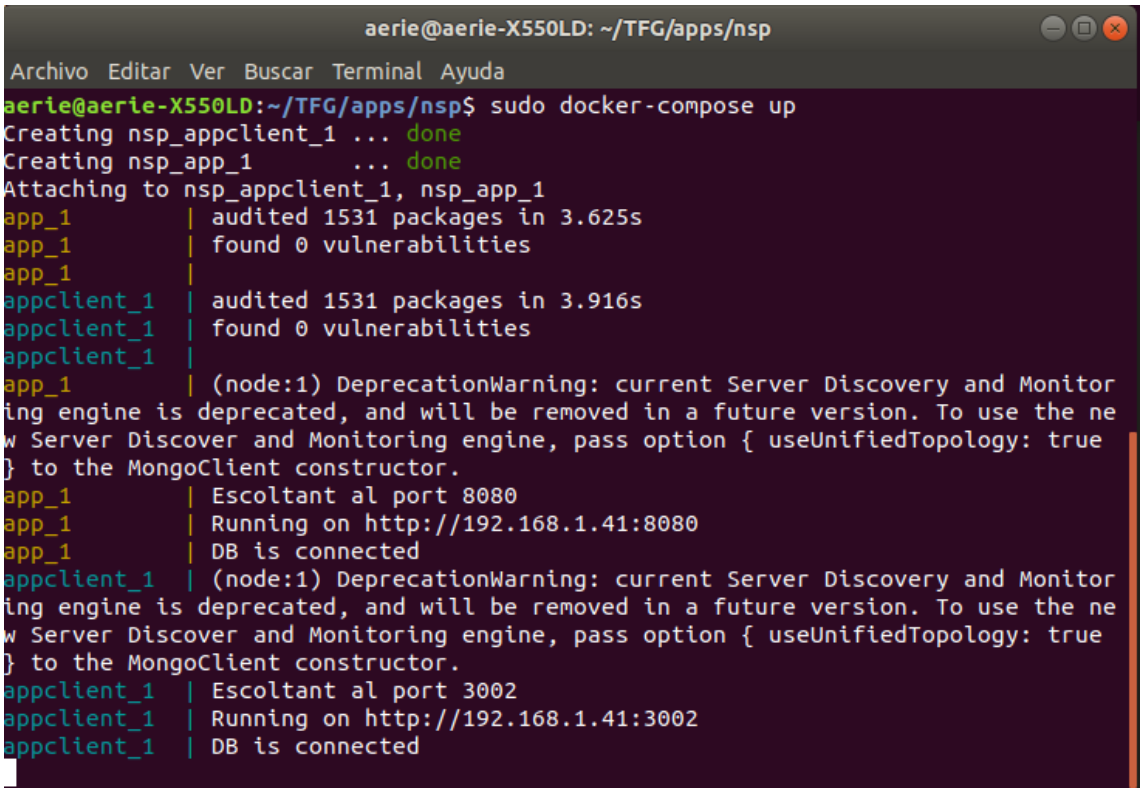


Figura 67: comprovació de que els contenidors estan en marxa i els servidors de les aplicacions estan corrent

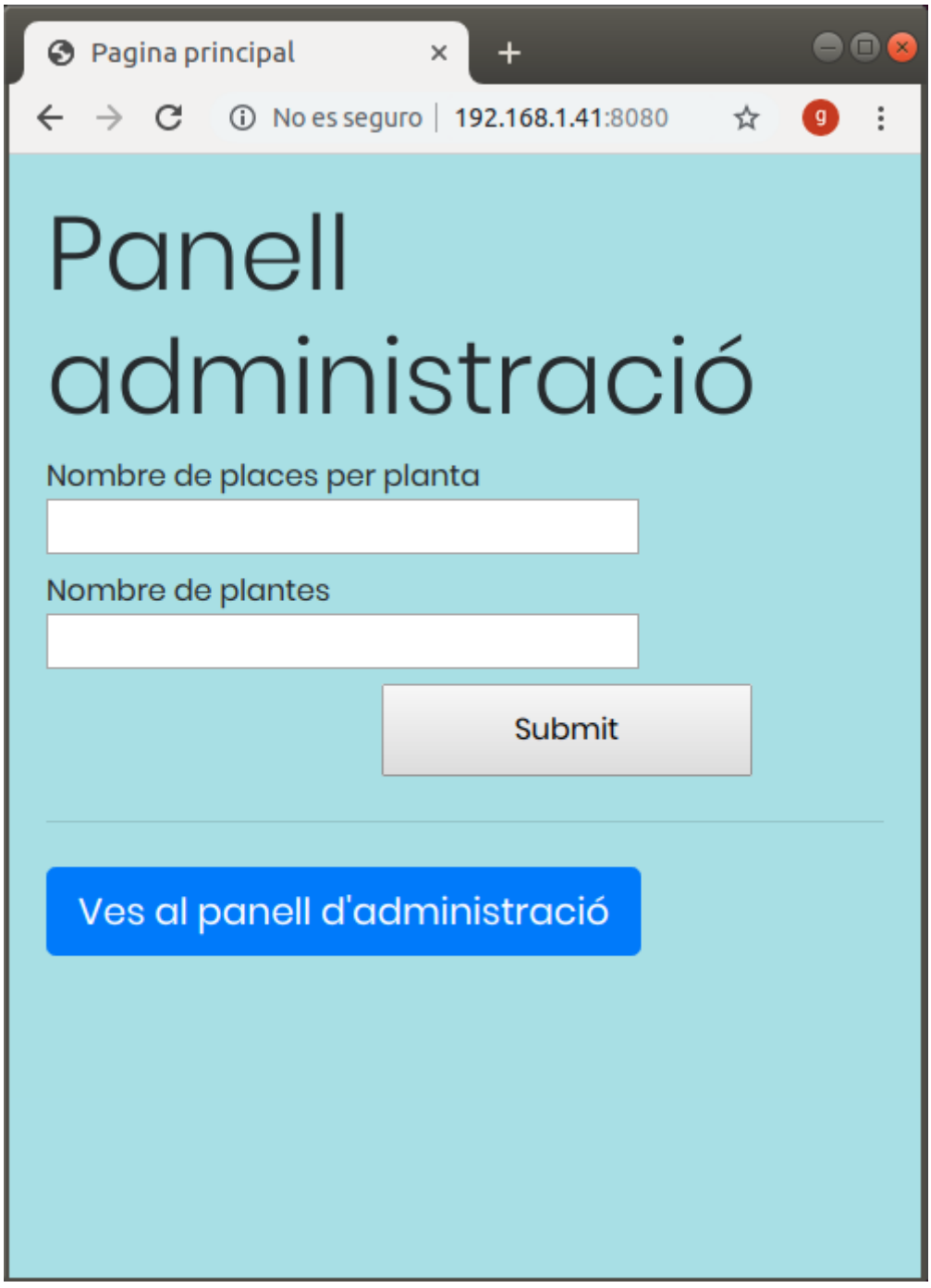


Figura 68: recurs servit correctament pel servidor de l'aplicació panell d'administrador

Veiem que les aplicacions estan escoltant a l'adreça IP 192.168.1.41 i als ports 8080 i 3002. També ens mostra un missatge confirmant que estan connectats a la base de dades MongoDB (DB is connected, figura 67). Si no tinguéssim el MongoDB corrent no es podria connectar i sortiria un error de connexió.

Ara, anem al navegador web i li demanem que ens serveixi els recurs localitzat a la ruta http://192.168.1.41:8080.

Fent això per a cada mòdul / ruta del front-end podem verificar que les peticions fetes des del client arriben al servidor i aquest li serveix correctament el contingut de la pàgina.

Ara demanarem que ens serveixi l'aplicació client i veurem com ens la mostra correctament (figura 69).

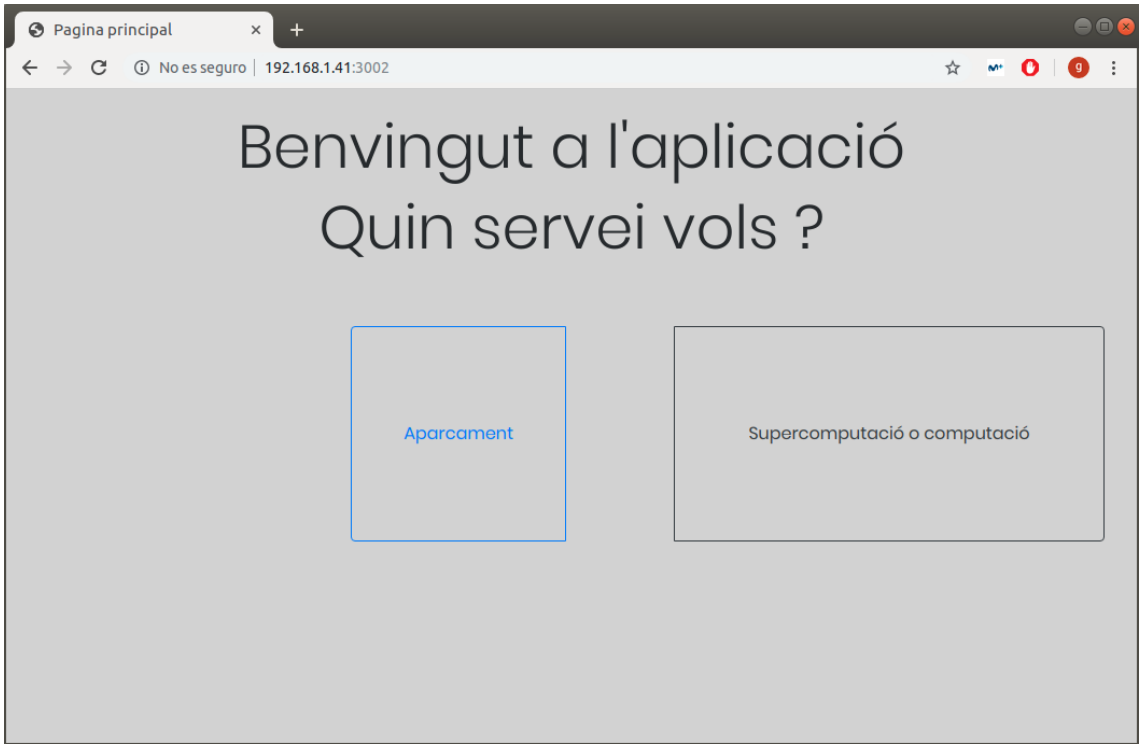


Figura 69: recurs servit correctament per el servidor de l'aplicació client

10.4 Test virtualització d'agents

Per a arrencar els contenidors de Docker haurem d'executar el fitxer /TFG/utils/agent_docker.py seguit del nombre d'agents que volem arrencar, tal i com mostra la següent imatge:

```
aerie@aerie-X550LD:~$ sudo python3 agent_docker.py 5
b3edaa136e09b9afb9b6e70c1160fb85aa86a1fc0b0b856103cba3bccde986f0
34bb5c70ca972b4e1df1748f876bea6c1159e12db81dedb27ed79c64a9321989
bb6d06d8ce6ba30e479198aff46ca4dd2432c882859d50043b1f8031a8aa9a65
8e9aa9c81191aea9be26bd568509db1bba2b8c571dbd9267c5c9a18114754c65
08df37887d9aecace3deb47b07a6a8885bed1605e1cf107fc18382c328249d5e
172.17.0.2

172.17.0.3

172.17.0.4

172.17.0.5

172.17.0.6
```

Figura 70: arrenquem 5 contenidors de Docker amb un programa Python

En aquest cas n'hem arrencat 5. Veiem com la sortida d'aquest script mostra per pantalla els identificadors dels contenidors de Docker creats i també les seves IP's. Després d'arrencar els contenidors, aquest programa també enregistra els vehicles al pàrquing. Així, si anem al panell d'administració veurem que s'han afegit 5 vehicles/agents al pàrquing.

10.5 Test de serveis

10.5.1 Servei de computació

Com ja s'ha comentat anteriorment, el servei de computació es sol·licita directament des del front-end, enviant un fitxer a computar.

Des d'un navegador anem a la ruta http://10.0.6.50:8080/computacio

Seleccionem el fitxer a computar.

Escollim la versió de Python i si fa ús de ParallelPython i fem clic a "Enviar" per enviar-lo a computar.

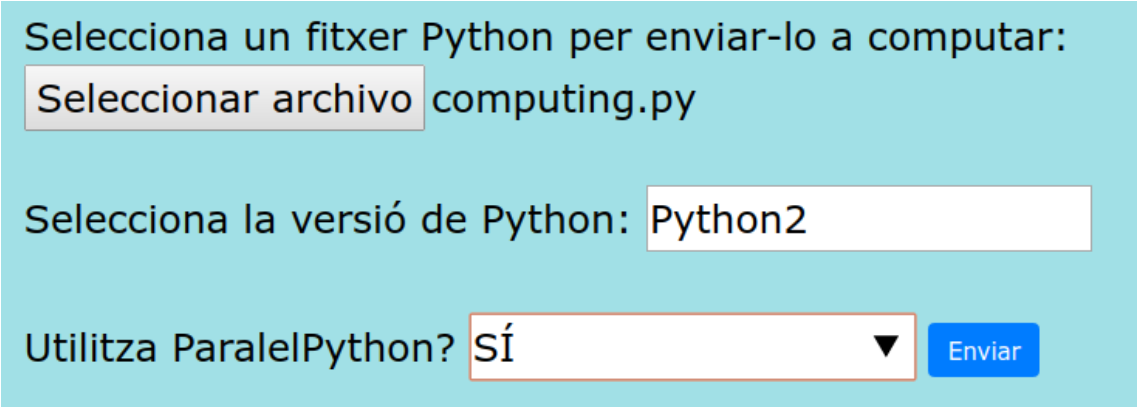
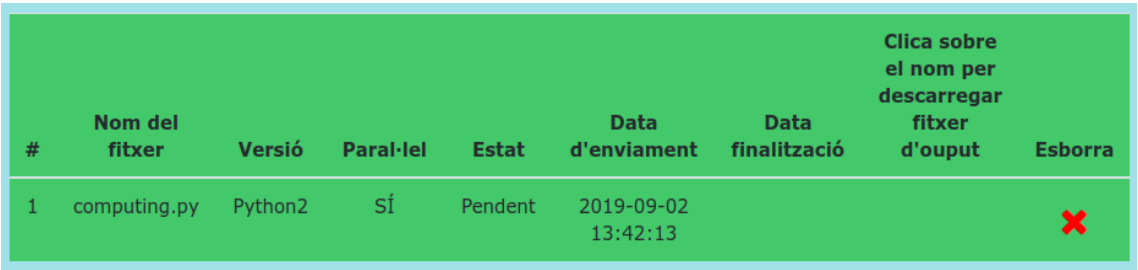


Figura 71: formulari a entrar pel client

Un cop enviat, haurem de comprovar com s'afegeix una nova fila a la taula corresponent al fitxer enviat.



#	Nom del fitxer	Versió	Paral·lel	Estat	Data d'enviament	Data finalització	Clica sobre el nom per descarregar fitxer d'ouput	Esborra
1	computing.py	Python2	SÍ	Pendent	2019-09-02 13:42:13			✖

Figura 72: la taula conté el fitxer i la seva informació un cop enviat a computar

Al finalitzar l'execució, veurem com s'omplen els dos camps restants i el camp "Estat" passa a ser finalitzat. Un cop executat la fila ha de quedar com segueix:

#	Nom del fitxer	Versió	Paral·lel	Estat	Data d'enviament	Data finalització	Clica sobre el nom per descarregar fitxer d'ouput	Esborra
1	computing.py	Python2	SÍ	Finalitzat	2019-09-02 13:42:13	2019-09-02 13:42:19	log-computing.txt	✖

Figura 73: fitxer acabat de computar

I finalment cliquem a sobre del nom del fitxer de sortida per a descarregar-lo:

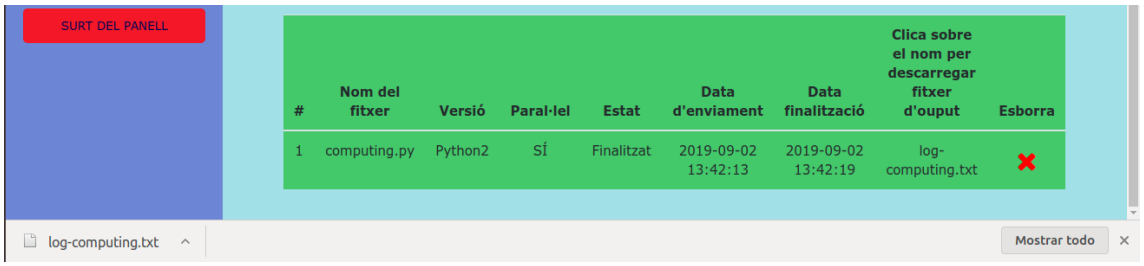


Figura 74: fitxer amb el resultat de l'execució del fitxer descarregat

Ja descarregat el fitxer amb el resultat de l'execució, podem esborrar el fitxer de la taula clicant a la creu vermella.

10.6 Proves de rendiment Parallel Python

Per tal de valorar el rendiment del nostre clúster de computació amb la llibreria Parallel Python s'han fet divereses proves. No obstant, algunes d'elles no s'han pogut dur a terme correctament per motius tècnics explicats més endavant en aquest mateix apartat.

S'ha executat el mateix programa amb diversos nombres de nodes per veure el temps d'execució. Aquest programa consta de 50.000 tasques molt petites que s'executen entre les diverses màquines virtuals. Cada tasca d'aquestes troba un cami més curt en un graf (és només un programa de proves, no fem res més amb ell).

La taula 4 mostra els resultats obtinguts:

Nº màquines virtuals	Temps d'execució*	Millora respecte 1 MV**
1	19.55014 segons	-
4	18.37195 segons	**6.02%** menys
6	17.95238 segons	**8.17%** menys
8	17.31781 segons	**11.41%** menys
12	18.71684 segons	**4.26%** menys

Taula 4: performance Parallel Python

* El programa ha sigut executat 5 vegades i s'ha fet la mitjana aritmètica de tots els temps, per tal de fer el testeig més fiable i més estable.

** MV significa Màquina Virtual. 1 màquina virtual correspon a l'execució seqüèncial (ja que no estem paral·lelitzant, s'executen totes les tasques en una sola màquina).

Com veiem (taula 4) arriba un punt en que el temps ja no millora. Això és deu a que estem corrent totes les màquines virtuals en un mateix ordinador portàtil, i quantes més en tenim més lent va el sistema en general. Per això triga més en fer-se l'execució amb 12 que amb 4.

Tot i així, podem dir que els resultats obtinguts són molt bons ja que només amb 7 màquines virtuals més guanyem un benefici de quasi un 11,5% menys de temps respecte a l'execució seqüèncial.

10.7 DEMO final

L'entrega final d'aquest projecte consisteix en l'aportació del present document juntament amb una representació pràctica amb la finalitat de posar a prova totes les funcionalitats implementades i verificar-ne el correcte funcionament davant del tribunal qualificador del projecte.

Així, en aquesta DEMO s'ensenyarà:

- Funcionament del panell d'administració
- Demostració del servei d'aparcament intel·ligent al testbed
- Demostració del servei de computació

CAPÍTOL 11 – CONTINUÏTAT DEL PROJECTE

En aquest projecte es presenten dues aplicacions de tipus web (el panell d'administració i l'aplicació per als clients per a demanar els serveis). Cap de les dues disposa de validació (log-in) per a l'usuari que l'està utilitzant. Això en un entorn real suposaria un greu forat de seguretat, ja que qualsevol persona podria entrar a l'aplicació només sapiguent a quina URL s'hi troba. És per això que una de les idees proposades com a treball futur seria la implementació d'un sistema d'enregistrament i validació (per exemple amb un parell usuari/contrasenya) a l'hora d'inicialitzar el front-end.

Una de les altres propostes seria el disseny i la implementació d'una aplicació mòbil (app) per a que els clients poguessin demanar els serveis des del seu telèfon mòbil. Amb una app mòbil resultaria molt més fàcil i còmode sol·licitar serveis ja que l'usuari només hauria de descarregar-la i obrir-la cada vegada que vulgui sol·licitar un servei (i no accedir a una URL amb el navegador, que per un usuari sense coneixements informàtics pot resultar més complicat).

Un altre aspecte a incorporar al projecte realitzat seria el pagament per part de l'usuari corresponent al temps que el seu vehicle ha estat estacionat al pàrquing, així com també la sortida del vehicle del pàrquing.

Per últim, es podria millorar el servei d'aparcament intel·ligent en el testbed, afegint nous elements al pàrquing del testbed, incorporant per exemple una barrera a l'entrada o afegir línies a cadascuna de les places per tal de que poguessim aparcar més d'un cotxe.

CAPÍTOL 12 – COST DEL PROJECTE

En aquest capítol estimarem el cost total que costaria posar en marxa el projecte en un entorn més real (no acadèmic).

El primer cost que ens trobem és el corresponent a les hores dedicades per el programador o equip de programadors encarregats del disseny del projecte i la codificació dels elements que el composen. Com ja s'ha detallat al capítol 2, han fet falta un total de 576 hores per al desenvolupament del projecte.

Atès que en el món de la informàtica hi ha diferents salaris depenent de la categoria i experiència del programador, he agafat la categoria "programador junior" (que és la que em correspondria a mi al finalitzar la carrera) i el seu sou base per al càlcul del cost.

Així, segons el BOE (Boletín Oficial del Estado) publicat al 2018, el sou base d'un programador junior a Espanya és de 14800€ bruts a l'any [34]. Llavors, assumint que cada mes té 20 dies laborables i es treballen de mitjana unes 8 hores al dia, tenim el següent càlcul:

14800€ / 12 mesos / 20 dies laborables / 8 hores al dia = 7.70€ aprox / hora

7.70€ / hora * 576 hores = 4440€

El segon cost correspon als servidors necessaris per a allotjar ("*hosting*") les aplicacions implementades (panell d'administrador i aplicació per al client) així com també les bases de dades.

Actualment existeixen moltes plataformes de *hosting* per a les aplicacions web. Una de les més populars i que ens podria servir per a les nostres aplicacions és Guebs [35]. Aquesta plataforma inclou suport per aplicacions que utilitzen Node.js i permet que aquestes accedeixin a bases de dades mongoDB. És per tant una plataforma adequada per al nostre projecte.

Existeixen diferents plans en funció de les necessitats del client. La que més ens convindria té un cost de 4,99€ al mes (per a cada aplicació) durant el primer any i posteriorment 9,99€ al mes.

Pel que fa a la part de software i eines utilitzades no requerim cap cost ja que totes són gratuïtes (Python, mongoDB, Docker, etc...)

Atès que els agents implementats són virtuals (i no físics), tampoc requereixen cap cost més enllà d'una màquina on córrer els contenidors de Docker.

Notem que en aquesta estimació de costs feta en aquest capítol no es tenen en compte elements bàsics i indispensables tals com el ordinador que el programador/s necessita per a desenvolupar el projecte o el testbed on s'ha provat el servei d'aparcament intel·ligent, ja que se suposa que són elements externs que l'empresa o client que vulgui realitzar el projecte ja té (i per tant no són costs propis del projecte).

CAPÍTOL 13 – CONCLUSIONS FINALS DEL PROJECTE

Un cop finalitzat el projecte podem afirmar de manera objectiva que tant els objectius tècnics com els objectius acadèmics descrits al capítol 1 han sigut assolits. A continuació es mostra la llista d'objectius inicials, juntament amb una breu explicació de com s'han assolit:

Objectius tècnics:

✓ Disseny i implementació a nivell de software dels agents, així com les comunicacions i pas de missatges entre ells: a partir del disseny modular acordat amb els clients s'han implementat els agent amb el llenguatge Python. Per a les comunicacions entre agents (entrada i sortida de dades) s'ha utilitzat una API REST implementada amb el framework web de Python anomenat CherryPy. Els agents compleixen amb les funcionalitats i requisits descrits al capítol 5, siguent l'execució de serveis la seva funcionalitat principal.

✓ Virtualització d'agents per tal de poder tenir-ne un nombre elevat (simulació d'una ciutat o un pàrquing real): s'han virtualitzat els agents amb Docker i s'ha fet un programa que arrenca n contenidors (n agents) de manera que en podem tenir un nombre elevat només executant una comanda.

✓ Disseny i implementació de serveis relacionats amb l'aparcament intel·ligent: com ja hem vist al llarg de la memòria, en aquest projecte s'han dissenyat i implementat dos serveis, partint sempre d'una arquitectura F2C: el servei de computació i el servei d'aparcament intel·ligent. El servei de computació permet a un client enviar a computar de manera distribuïda (o no) un fitxer al clúster d'agents del pàrquing. El servei d'aparcament intel·ligent permet a un client aparcar el seu vehicle de manera eficaç, eficient i sencilla.

✓ Integració de part del projecte amb altres projectes que s'estan desenvolupant paral·lelament a aquest per tal de potenciar algunes de les funcionalitats implementades: tal i com es detalla al capítol 9, s'ha integrat el servei d'aparcament intel·ligent fet en aquest projecte amb un altre projecte que tracta de conducció autònoma. S'ha aconseguit fer la integració amb èxit i executar el servei d'aparcament al testbed del CRAAX.

✓ Realització de les proves pertinents per verificar el correcte funcionament de cada element implementat en el projecte: com es mostra al capítol 10, s'han portat a terme una sèrie de proves que han validat el projecte i cadascun dels seus elements.

✓ Disseny i implementació d'una aplicació web (front-end) per administrar un pàrquing F2C: gràcies a l'aplicació realitzada l'usuari administrador pot veure quins fitxers s'estan computant en el pàrquing, veure quins vehicles hi han estacionat i la seva informació (matrícula, recursos computacionals, etc) i altres dades d'interés (reserves actuals, quantitat total de recursos disponibles, etc).

✓ Disseny i implementació d'una aplicació destinada als nostres clients per tal de que puguin sol·licitar els serveis implementats en el projecte: s'ha dissenyat i implementat una petita aplicació web capaç de sol·licitar els dos serveis implementats d'una manera sencilla, ràpida i molt visual.

✓ Elecció i aprenentatge autònom de les tecnologies pertinents per a desenvolupar el projecte: investigació i elecció de les tecnologies i eines que millor s'adapten a les característiques del projecte per posteriorment apendre-les i poder desenvolupar el projecte.

Objectius acadèmics:

✓ Aplicació d'una metodologia àgil per a desenvolupar el projecte: un altre aspecte a valorar és la metodologia seguida (variant de SCRUM per a equips d'un sol desenvolupador). A cada sprint s'ha dut a terme un increment del projecte i s'ha presentat el resultat al client fins arribar a la realització completa del projecte. S'ha fet un ús correcte de *Trello* per administrar i organitzar les tasques a fer i per a la comunicació amb els clients.

✓ Treballar paral·lelament i conjuntament amb altres projectes que estan sent desenvolupats per altres estudiants del mateix grau i enfocats en l'àmbit de ciutats intel·ligents: podem afirmar que aquest objectiu s'ha assolit ja que hi han hagut reunions i quedades continuament amb els altres equips per tal de consensuar com implementarem el servei d'aparcament conjunt. A més, totes les demos s'han fet amb els altres equips i per tant cada equip sap de que van els altres projectes.

✓ Assentar les bases d'una futura línia de treball en termes d'aparcament intel·ligent i ciutats intel·ligents: gràcies a la realització d'aquest projecte el CRAAX disposa d'una base per al les seves investigacions en l'àmbit d'aparcament intel·ligent dins de ciutats intel·ligents. Aquest projecte, doncs, podrà ser el punt de partida d'altres projectes que vulguin millorar el sistema d'aparcament intel·ligent proposat o vulguin utilitzar-lo per algun altre fi.

✓ Redactat de la present memòria i documentació tècnica (funcionament de cada element implementat i manuals per ficar-ho tot en funcionament): la present memòria ha sigut revisada i validada per part del tutor del projecte. De la mateixa manera, el projecte ha sigut instal·lat satisfactòriament a la màquina del client seguint el manual d'instal·lació, detallat a l'annex 1.

✓ Aconseguir la resta d'objectius plantejats per tal d'oferir al CRAAX el resultat final del projecte: Finalment observem com s'han assolit tots els objectius inicials, fet que ha permés entregar el projecte al CRAAX per tal de que puguin utilitzar-lo i afegir-li millores en un futur.

Com a conclusió final del projecte podem dir que amb la realització d'aquest projecte s'ha aconseguit assentar les bases d'una futura línia de treball en l'àmbit d'aparcament intel·ligent, monitoritzar l'estat d'un pàrquing mitjançant un panell d'administració i finalment s'ha aconseguit l'execució de serveis relacionats amb l'aparcament intel·ligent, partint sempre d'una arquitectura F2C.

13.1 Valoració personal

A nivell personal la realització d'aquest projecte ha sigut una experiència satisfactòria i enriquidora. He après tecnologies i llenguatges de programació que de ben segur em seran d'utilitat en el futur doncs es tracten de llenguatges i tecnologies punteres molt utilitzades en el món laboral.

La metodologia emprada i el fet de poguer anar al laboratori del CRAAX a desenvolupar el projecte ha sigut de gran ajut ja que d'aquesta manera el contacte amb el client és continu i això ha ajudat a resoldre els dubtes que anaven sorgint.

La principal dificultat que m'he trobat a l'hora de realitzar el projecte (sobre tot al principi) han sigut les pròpies de realitzar un projecte d'aquestes característiques, és a dir, treballar en un projecte d'investigació on al principi no estaven del tot definits els objectius i es va haver de pensar què es faria i de quina manera.

CAPÍTOL 14 – Referències

[0] https://www.researchgate.net/figure/Internet-of-Things-IoT-connected-devices-from-2015-to-2025-in-billions_fig1_325645304

[1] Centre de Recerca d'Arquitectures Avançades de Xarxes. CRAAX: https://craax.upc.edu/

[2] Escola Politècnica Superior d'Enginyeria de Vilanova i la Geltrú. EPSEVG: https://www.epsevg.upc.edu/ca

[3] Edifici Neàpolis de Vilanova i la Geltrú : http://www.neapolis.cat/

[4] Trello: https://trello.com

[5] Empresa analista Gartner: https://www.gartner.com/en

[6] Diari britànic "The Guardian": https://www.theguardian.com/international

[7] Hadoop: https://hadoop.apache.org/

[8] Spark: https://spark.apache.org/

[9] Smart Santander: http://www.smartsantander.eu/

[10] Waspmotes: http://www.libelium.com/products/waspmote/

[11] Arduino: https://www.arduino.cc/

[12] Parking Callao: https://www.parkingcallao.es/

[13] Mobypark: https://www.mobypark.com/es

[14] Cisco: https://www.cisco.com/

[15] mF2C: https://www.mf2c-project.eu/

[16] Hospital Clínic de Barcelona: https://www.clinicbarcelona.org/ca

[17] Raspberry Pi: https://www.raspberrypi.org/

[18] MongoDB: https://www.mongodb.com/

[19] MySQL: https://www.mysql.com/

[20] Node.js: https://nodejs.org/en/

[21] Google: https://ca.wikipedia.org/wiki/Google

[22] Express framework: https://expressjs.com/

[23] Sublime Text: https://www.sublimetext.com/

[24] Postman: https://www.getpostman.com/

[25] Oracle Corporation: https://www.oracle.com/corporate/

[26] VirtualBox: https://www.virtualbox.org/

[27] Sistema operatiu Debian: https://www.debian.org/index.ca.html

[28] Paralel Python: https://www.parallelpython.com/

[29] MongoDB Compass: https://www.mongodb.com/products/compass

[30] Draw.io: https://www.draw.io/

[31] Protocol SSH: https://ca.wikipedia.org/wiki/Secure_Shell

[32] Docker: https://www.docker.com/

[33] Google Chrome: https://www.google.com/intl/ca_ES/chrome/

[34] BOE – Sou base d'un programador junior a Espanya
https://www.boe.es/boe/dias/2018/03/06/pdfs/BOE-A-2018-3156.pdf

[35] Guebs: https://www.guebs.com/hosting/nodejs

CAPÍTOL 15 - Bibliografia

Introducció a l'element CANVAS d'HTML 5

- https://www.w3schools.com/html/html5_canvas.asp

Tutorial Docker Node.JS

- https://nodejs.org/de/docs/guides/nodejs-docker-webapp/

Projecte mF2C

- https://www.mf2c-project.eu/

Llibreria Parallel Python

- https://www.parallelpython.com

CRAAX

- https://www.craax.upc.edu/index.php/about-us

Informació F2C

- https://www.sam-solutions.com/blog/fog-computing-vs-cloud-computing-for-iot-projects/
- https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf
- https://www.gradiant.org/blog/edge-fog-computing-cloud/

Informació IoT:

- https://iurban.es/la-importancia-del-iot-para-las-smart-cities/

- https://www.forbes.com/sites/jacobmorgan/2014/05/13/simple-explanation-internet-things-that-anyone-can-understand/#5c64c6ee1d09

Informació smart cities

- http://www.panelesach.com/blog/smart-cities-o-ciudades-inteligentes-que-son/
- https://www.esmartcity.es/medio-ambiente
- https://iurban.es/que-hacen-las-smarts-cities-mas-exitosas-del-mundo/
- http://www.smartsantander.eu/index.php/testbeds/item/132-santander-summary

Informació smart parking

- https://www.convi.net/los-10-beneficios-del-smart-parking/
- https://www.convi.net/que-es-el-smart-parking/
- http://www.libelium.com/smart_santander_smart_parking/

Exemples de la llibreria Chart.js

- https://www.chartjs.org/samples/latest/

Model-Vista-Controlador

- https://ca.wikipedia.org/wiki/Model-Vista-Controlador

Estadístiques Big Data:

- https://techjury.net/stats-about/big-data-statistics/
- https://www.theguardian.com/news/datablog/2012/dec/19/big-data-study-digital-universe-global-volume

ANNEX 1 – MANUALS

Aquest annex pretén ser una guia d'instal·lació i posada en marxa del present projecte, així com un petit manual de com executar un programa amb la llibreria Parallel Python i de com utilitzar la API implementada.

Prerequisits

Encara que suposem que són prerequisits necessaris a continuació es deixen les comandes per a instal·lar cadascun d'ells i el link d'on s'han tret:

1. Entorn **Linux** (Ubuntu preferiblement)

2. **Python2** + **pip**

   $ sudo apt install python

   $ sudo apt install python-pip

3. **Python3** (instal·lat de sèrie amb les últimes versions d'Ubuntu) + **pip**

   $ sudo apt install python3

   $ sudo apt install python3-pip

4. **Docker + docker-compose**

Docker (https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-18-04)

$ sudo apt update

$ sudo apt install apt-transport-https ca-certificates curl software-properties-common

$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu bionic stable"

$ sudo apt update

$ apt-cache policy docker-ce

$ sudo apt install docker-ce

Docker-compose (https://docs.docker.com/compose/install/)

$ sudo curl -L "https://github.com/docker/compose/releases/download/1.24.1/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

$ sudo chmod +x /usr/local/bin/docker-compose

5. **MongoDB** (https://www.digitalocean.com/community/tutorials/how-to-install-mongodb-on-ubuntu-18-04)

$ sudo apt install mongodb

Guia d'instal·lació del projecte

Pas 1: descarregar codi font del projecte i descomprimir-lo **en el directori personal de l'usuari**. Per exemple, si el nom de l'usuari és nil, descomprimirem en el directori /home/nil/, de manera que ens quedi com a /home/nil/TFG/...

https://github.com/Nil22/TFG

Important canviar el nom del directori arrel a TFG i descomprimir-lo en el directori adequat sinó no funcionaran alguns aspectes del projecte per tema dels "paths" o camins dins dels directoris de la màquina.

Pas 2: instal·lar Parallel Python i llibreries dels agents

```
$ pip install pp
```

```
$ cd /TFG/
```

```
$ pip3 install -r requirements.txt
```

El fitxer requirements.txt conté les dependències que la classe agent necessita per a ser executada amb èxit. Parallel Python funciona amb la versió 2 de Python i per tant necessita ser instal·lat amb pip.

Pas 3: executar el fitxer /utils/dbconfig.py

```
$ cd /TFG/utils
```

```
$ sudo python3 dbconfig.py
```

Aquest fitxer ens crea les bases de dades necessàries al mongoDB i ens introdueix els elements necessaris per a posteriorment fer-hi les operacions. No passa res si veiem un error durant l'execució, es produeix si ja tenim el mongoDB corrent al sistema.

Pas 4: instal·lar npm

```
$ sudo apt update

$ sudo apt install npm
```

Pas 5: configuració servidor SFTP

Per a configurar la connexió amb el directori que farà de SFTP obrim amb un editor de text el fitxer /TFG/agents/sex.py

Anem a la funció get_code(self, serviceID) (la darrera funció de totes) i modifiquem el path on tenim el directori SFTP, (línia 61 en el codi) que serà del tipus /home/<nom_usuari>/TFG/SFTP/...

Ara, canviem els valors "username" i "password" (línia 65 en el codi) per al nom del nostre usuari i la seva contrasenya en el sistema.

Nota: la llibreria Pysftp no agafa paths relatius i per tant en aquest cas la configuració s'ha de fer manualment. El mateix passa amb l'usuari i contrasenya.

Arribats a aquest punt ja tenim el sistema preparat amb tot el programari i llibreries que es necessiten.

Posada en marxa del el projecte

Pas 1: arrencar MongoDB

```
$ sudo mongod --fork --logpath /var/log/mongod.log
```

Si a l'executar la comanda anterior ens dona error, vol dir que MongoDB ja està corrent. Ho comprovem amb l'ordre $ mongo i veiem com ens entra a la consola de MongoDB. Això es deu a que al instal·lar mongoDB ja corre automàtiament, si no hem apagat el PC (o a vegades ja està configurat per defecte per arrencar-lo a l'iniciar el sistema).

Pas 2: arrencar agent leader del parquing

```
$ cd /TFG/agents

$ sudo python3 main_agent.py 8000
```

On 8000 és el port on escoltarà la API de l'agent. La API escoltarà per defecte a l'adreça IP 0.0.0.0 i al port 8000. Per canviar la funció o la informació de l'agent que estem iniciant haurem d'entrar al codi del fitxer main_agent.py i canviar els paràmetres per a la creació de l'objecte agent.

Pas 3: arrencar apps front-end

```
$ cd /TFG/apps/nsp

$ sudo docker-compose up
```

Amb la darrera comanda arrencarem 2 contenidors Docker amb la imatge de Node.js que corren les aplicacions panell d'administració i l'aplicació per al client. Quan la comanda s'hagi executat, podrem accedir a dites aplicacions des d'un navegador web i utilitzar-les. Per a veure a quina IP i port ha arrencat els front-ends ens podem fixar en els missatges que surten quan executem la comanda.

Pas 4 (opcional): arrencar agents virtualitzats

Un pas previ a aquest pas seria descarregar del següent enllaç la imatge amb la qual s'han virtualitzat els agents. Aquesta imatge (ubuntu) té instal·lades les llibreries necessàries per als agents.

https://drive.google.com/file/d/1MG4ELjGgPGt4Nc2pSr-3oB6rqhEycwTy/view

Un cop descarregat el fitxer .tar.gz, l'importem fent:

```
$ sudo docker load < imatgeAgent.tar.gz
```

I posteriorment

```
$ cd /TFG/utils
```

```
$ sudo python3 agent_docker.py 5
```

Ens arrencaria 5 contenidors de Docker (agents virtualitzats) amb la imatge importada i els enregistraria al leader del al pàrquing (notem que aquest ha d'estar actiu per a que la seva API pugui atendre les peticions).

```
aerie@aerie-X550LD:~/Documentos/DEMO10/shar$ sudo python3 main_newagent.py 8000
[02/Sep/2019:12:50:22] ENGINE Listening for SIGTERM.
[02/Sep/2019:12:50:22] ENGINE Listening for SIGHUP.
[02/Sep/2019:12:50:22] ENGINE Listening for SIGUSR1.
[02/Sep/2019:12:50:22] ENGINE Bus STARTING
CherryPy Checker:
The config entry 'tools.response.headers.headers' may be invalid, because the 'r
esponse' tool was not found.
section: [/]

[02/Sep/2019:12:50:22] ENGINE Started monitor thread 'Autoreloader'.
[02/Sep/2019:12:50:22] ENGINE Serving on http://10.0.6.50:8000
[02/Sep/2019:12:50:22] ENGINE Bus STARTED
```

Figura 75: execució satisfactòria d'un agent amb la API escoltant a la IP 10.0.6.50 i port 8000

Com executar un programa amb la llibreria ParallelPython

El primer que hem de fer és veure quins nodes tenim actius a la xarxa. És a dir, en quines màquines s'executarà el nostre programa.
Amb la línia de codi següent declarem una variable que les conté totes (figura 75):

```
# autodiscovery
ppservers = ("*",)  # the comma is important!
```

Figura 76: descobriment automàtic dels nodes capaços de computar

Si no volem que es fagi un autodescobriment per detectar les màquines també podem indicar-li les IP de cada màquina, separades per comes.
Un cop tenim els nodes crearem el que serà la tasca principal (servidor) i li indiquem els nodes que s'han trobat (figura 76).

```
# crea la tasca principal amb els nodes trobats
job_server = pp.Server(ppservers=ppservers)
```

Figura 77: crea el servidor amb els nodes trobats a la xarxa

Ara ja podem crear les diverses tasques que s'executaran en cadascun dels nodes computacionals de la xarxa.

Per crear una tasca fem:

```
job1 = job_server.submit(calcula_distancia, (500,500,250,250,))
result = job1()
print result
```

Figura 78: creació d'una tasca amb Parallel Python

Aquí (figura 77) estem creant una tasca job1 i enviant-la per a que s'executi. calcula_distància és el nom d'una funció que tenim definida al mateix fitxer, i els quatre nombres corresponen als paràmetres que rep la funció.
Per obtenir el resultat de l'execució d'aquesta tasca només cal fer job1().
Notem que el resultat de la tasca serà del mateix tipus que el que ens retorna la funció calcula_distància. Per exemple, si calcula_distancia fa una suma dels quatre nombres que li passem com a paràmetres, la variable result serà igual a 1500.

Què passa si tenim diverses tasques a executar i són tasques dependents?

En cas de tenir una tasca que depèn d'una anterior a ella, haurem de esperar a que la tasca anterior acabi la seva execució, per posteriorment poder utilitzar els valors obtinguts.

```
if job1.finished:
    job2 = job_server.submit(calcula_distancia, (result,result,result,result,))
    result2 = job2()
    print result2
```

Figura 79: comprovació de que una tasca ha finalitzat

En el codi de la imatge anterior, es comprova que la tasca «job1» hagi finalitzat, i si ha finalitzat s'utilitza el seu resultat (variable result en el codi) com a paràmetre de la tasca «job2». Ens hem d'assegurar de que la primera tasca hagi acabat ja que sinó la variable result tindrà un valor erroni.

Quan haguem acabat de definir totes les tasques, podem utilitzar la següent funció (figura 79) per veure les tasques que s'han executat a cada màquina o el temps d'execució.

```
job_server.print_stats()
```

Figura 80: mostra les estadístiques de l'execució amb Parallel Python

Què passa si enmig de l'execució d'una tasca cau el node que s'encarregava d'executar-la?

La llibreria ParallelPython incorpora un mecanisme que fa un *rescheduling* (reprogramació de les tasques) en el cas de que una tasca es comenci a executar en un node però per algun motiu no acabi l'execució.
De manera que no ens haurem de preocupar per això.

API

A continuació es mostren algunes de les trucades més importants de la API implementada per als agents.

Request: **GET** /get_agents

Response: array de diccionaris on cada diccionari conté la informació d'un agent de la base de dades topològica.

Request: **GET** /get_leader_agents/<ip leader>

Response: array de diccionaris on cada diccionari conté la informació d'un agent connectat al leader amb IP = <ip leader>.

Request: **GET** /get_vehicles

Response: array de diccionaris on cada diccionari conté la informació d'un agent d'un vehicle estacionat al pàrquing i informació del vehicle.

Request: **GET** /get_vehicle/<nom de la plaça>

Response: diccionari amb la informació del vehicle aparcat a la plaça amb nom = <nom de la plaça>

Request: **GET** /get_service/<service ID>

Response: diccionari amb la informació del servei amb identificador = <service ID>

Request: **GET** /algorisme

Response: nom de la següent plaça disponible al pàrquing. Quan un vehicle entra al pàrquing fem aquesta petició per a saber a quina plaça ha d'aparcar.

Request: **POST** /register_agent

Body: {

"myIP" : "IP de l'agent",

"port" : "port on escolta la API de l'agent",

"leaderIP" : "IP del seu leader",

"IoT" : "dispositius IoT associats,

"role" : "funció de l'agent",

"device" : "tipus de dispositiu"

}

Response: identificador de l'agent amb el qual s'ha registrat a la base de dades topològica.

Request: **POST** /execute_service

Body: {

"service_id" : "<id del servei>",

"agent_ip" : <"ip del agent sol·licitant">

}

Response: execució del servei amb identificador <id del servei>.